MCUeZ

Easy development software from the company that knows MCU hardware best

MCUez HC12 Debugger User's Manual
MCUEZDBG12/D
Rev. 1

MOTOROLA

MCUEZDBG12/D
Rev. 1

MCUez HC12 Debugger

User's Manual

MOTOROLA

Important Notice to Users

Trademarks

List of Sections

List of Sections

Table of Contents

Section 6. D-Bug12 Monitor Target Component

Section 7. FLASH Programming

List of Figures

List of Figures

List of Figures

List of Tables

Section 1. General Information

1.1 Contents

1.2 Introduction

Motorola’s MCUez debugger incorporates a powerful GUI (graphical user interface) and command line that enables the user to debug assembly files, correlate them with the data, manipulate register contents, and read and manipulate memory contents.

Additional functionality for C source-level debugging is available with the corresponding license key. Refer to **Appendix C. Extended Backus-Naur Form (EBNF)** for additional information.

This manual describes how to use the debugger application. This chapter presents information on documentation conventions and provides a functional description of operation. The manual is divided into seven chapters, four appendices, and an index:

- **Section 1. General Information** provides a functional description of operation and support information.
- **Section 2. Graphical User Interface (GUI)** provides information on the toolbar, status bar, object information bar, drag and drop capability, and the debugger menus.
- **Section 3. Component Windows** provides detailed information on each basic debugger window component.
- **Section 4. Operating Procedures** contains procedures on how to use the MCUez debugger.
- **Section 5. Command Reference** provides detailed information on all MCUez commands. Each description also includes a usage example.
- **Section 6. D-Bug12 Monitor Target Component** provides information as it relates to the MCUez debugger environment.
- **Section 7. FLASH Programming** provides information on controlling on-chip FLASH devices.
- **Appendix A. Register Description File** provides definitions of the I/O registers used when loading a MCUez target.
- **Appendix B. C Source-Level Debugging** provides information on how to use the debugger when debugging C code at the source level.
- **Appendix C. Extended Backus-Naur Form (EBNF)** describes file formats and syntax rules.
- **Appendix D. Serial Device Interface (SDI)** provides information on the serial device interface.

1.3 Document Conventions

This section describes terms and styles used throughout the manual.

1.3.1 General Term

The following general term is used in this document:

Key1 + Key2

The + (plus) sign means that Key1 is held down while Key2 is pressed.

1.3.2 Mouse Operations

This bulleted list describes the terminology used in this manual to define mouse operations.

- Click — Implies to click the left mouse button once
- Right click — This click operation is done with the right mouse button.
- Double click — This indicates to double click the left mouse button.
- Drag — Press and hold down the left mouse button while dragging the mouse. The object will move with the mouse cursor and drop when the mouse button is released.
- Unclick — Release the mouse button.

1.3.3 Typographic Styles in This Manual

These typographic conventions are used in this manual:

- **Bold face** type is used for literal strings that must be used exactly as shown in the example and for the names of menus, windows, dialog boxes, icons, and buttons.
- `Courier` type face is used for all C-code program listings, command lines, and directories.
- *Italics* are used where the string is a place holder that may be substituted for a string of the user's own design.

- Variable user inputs are in *`Courier`* italics.
- Filenames are in italics with all lower case letters, for example, *proj.ext.*

These styles are used in this manual to define notational conventions:

- **Numeric constants —** Numeric constants are displayed in the C language format. Constants that are in the 0x format are hexadecimal. Constants that have no prefix are assumed to be decimal. The notation k, unless to denote a frequency setting in kilohertz, defines a number multiplied by 1024.
- **Function prototypes —** Structures and function call descriptions are given in terms of the C language. This does not limit the implementation of calling programs to C, but it is the calling routine's responsibility to provide the correct link to these routines.

1.4 Functional Description

The MCUez debugger is a multipurpose tool used for various tasks in the embedded systems and industrial control world. Some typical tasks are:

- Emulation and/or cross-debugging of an embedded application or hardware design using a graphical user interface
- Building a target application using an object-oriented approach

The MCUez debugger consists of the engine and a set of subwindow components bound to the task they perform (for example, a debugging session). The debugger engine is the heart of the system. It monitors and coordinates the tasks of the component windows. Each component window provides a separate function.

1.5 Component Windows

A component window can be inserted or removed from the debugger main window. Component windows are titled:

- Assembly
- Command
- Data
- Memory
- Module
- Register
- Source

Section 2. Graphical User Interface (GUI)

2.1 Contents

2.2 Introduction

All components are accessed from the MCUez debugger main window. The main window provides a menu bar, toolbar, and status information bar.

The main window manages the layout of the different component windows. The component windows are organized as follows:

- Multiple windows can be tiled or cascaded in the debugger window.
- Component windows are automatically resized when the main window is resized.
- Windows can be overlapped.
- Windows can be minimized.
- Several windows for the same component can be open.

This chapter provides information on the:

- Toolbar
- Status bar
- Component window object information bar
- Drag and drop among component windows
- Menus

2.3 Toolbar

A brief description appears on the screen when the mouse pointer is pointed at an icon. The toolbar is illustrated in **Figure 2-1**.

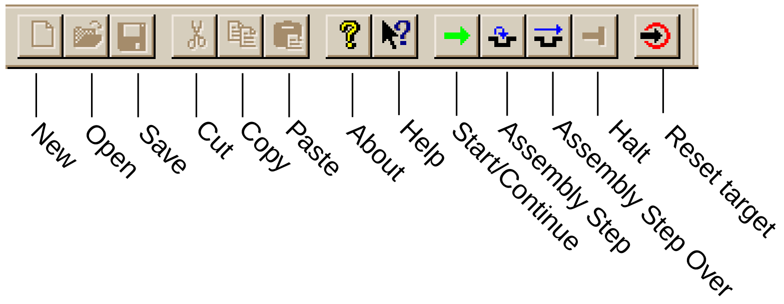


Figure 2-1. MCUez Debugger Toolbar

2.4 Status Bar

The status bar at the bottom of the debugger window contains a help line that displays a brief explanation when the mouse cursor is positioned over a button or menu item. Also shown is target-specific information. See **Figure 2-2**.

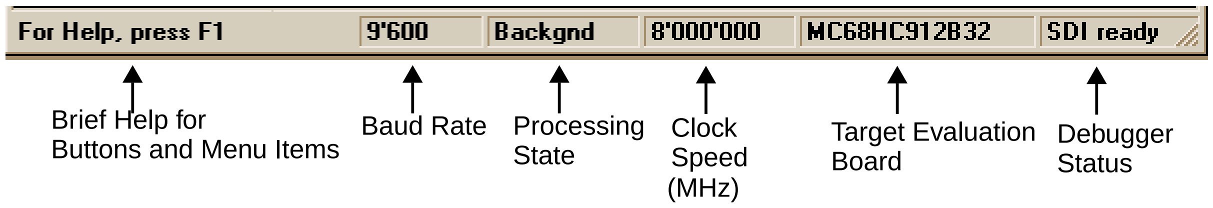


Figure 2-2. MCUez Debugger Status Bar

2.4.1 Debugger Status

SDI Ready	The debugger is ready and waits until a new target or application is loaded. This message is generated once the debugger has been started.
Halt	The application has been stopped by a request from the application.
Running	The application currently is executing in the debugger.
Halted	The application has been stopped by the user. The menu entry **Run | Halt** or the **HALT** icon in the toolbar has been selected.
Single_Step	The application stops after a single step through the source code. The menu entry **Run | Assembly Step** or the **Assembly Step** icon in the toolbar initiates a step.
Traced	Displayed after subsequent steps through code or when **Assembly Step Over** has been initiated
Breakpoint	Indicates that application has stopped at a defined breakpoint

2.4.2 MCU Error Messages

Some error messages depend on the MCU being used. These messages are related to exceptions. The debugger makes a distinction between predefined exceptions and user-defined exceptions. A user-defined exception has this format:

Exception `<string>|<number>`

Where:

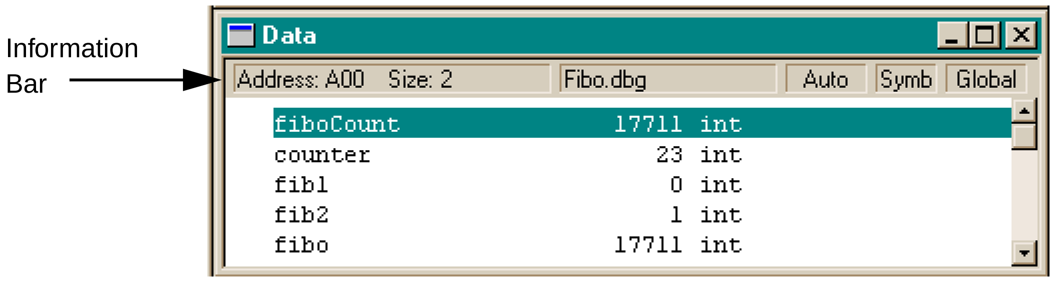
- `string` describes the reason for the exception. This string is only specified when a predefined exception is detected.
- `number` is the entry in the vector table that generates the exception. This number is only specified when a user-defined exception is detected.

The address error and bus error exceptions are treated differently.

Address Error	Indicates that an address error exception for the target processor has been generated. Check the hardware manual for the reason of the address error exception.
Bus Error	Indicates that a bus error exception for the target processor has been generated. Check the hardware manual for the reason of the bus error exception.
Other Exceptions	An exception has been generated for a vector that is not associated with an interrupt function. Possible reasons are: • Interrupt source was not disabled. • Entry in vector table that corresponds with the address of the function associated with interrupt was not initialized.

2.5 Information Bar

The information bar provides information about an item selected in the **Data** component window. See **Figure 2-3**.

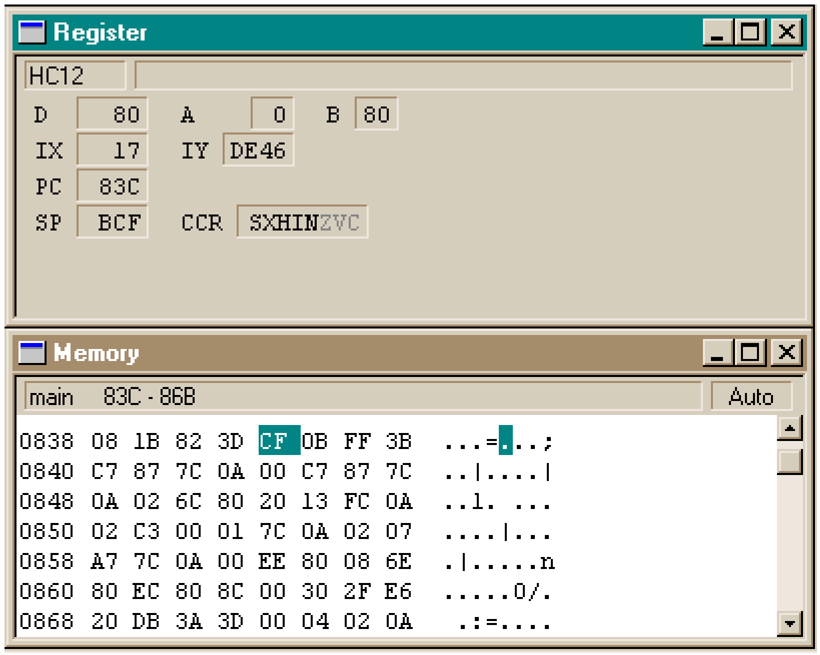


```
fiboCount                17711  int
counter                     23  int
fib1                         0  int
fib2                         1  int
fibo                     17711  int
```

Figure 2-3. Information Bar

2.6 Drag and Drop

The user can drag objects from one component window to another. This is defined as "drag and drop." For example, the user can display the memory layout corresponding to the address held in a register by dragging the address from the register component to the memory component. See the example in **Figure 2-4**.



```
D    80    A      0    B  80
IX   17    IY  DE46
PC  83C
SP  BCF    CCR SXHINZVC
```

```
0838  08 1B 82 3D CF 0B FF 3B   ...=...;
0840  C7 87 7C 0A 00 C7 87 7C   ..|....|
0848  0A 02 6C 80 20 13 FC 0A   ..l. ...
0850  02 C3 00 01 7C 0A 02 07   ....|...
0858  A7 7C 0A 00 EE 80 08 6E   .|.....n
0860  80 EC 80 8C 00 30 2F E6   .....0/.
0868  20 DB 3A 3D 00 04 02 0A    .:=....
```

Figure 2-4. Drag and Drop Example

To perform drag and drop operations:

1. Select the component window containing the object to drag.
2. Ensure that the destination component window is visible.
3. Select the object with the left mouse button and hold down the button.
4. Drag the object into the destination component window and release the mouse button.

The following sections describe the possible drag and drop combinations between component windows and the resulting operations.

If the destination of a dragged item is not possible, the cursor is displayed as a circle with a line through it.

2.6.1 Dragging from the Assembly Component

Drag and drop actions shown in **Table 2-1** are possible from the assembly component.

Table 2-1. Dragging from the Assembly Component

Destination Component	Operation
Command Line	The command line component appends the address of the pointed to instruction to the current command.
Memory	Dumps memory starting at the selected instruction program counter (PC). The PC location is selected in the memory component.
Register	Loads the destination register with the address of the selected instruction
Source	Source component scrolls to the corresponding source statement and highlights it.

2.6.2 Dragging into the Assembly Component

The events shown in **Table 2-2** occur when dragging and dropping into the assembly component.

Table 2-2. Dragging into the Assembly Component

Source Component	Operation
Source	Displays disassembled instructions starting at the first high level language instruction selected. The assembler instructions corresponding to the selected high level language instructions are highlighted in the assembly component.
Memory	Displays disassembled instructions starting at the first address selected. Instructions corresponding to the selected memory area are highlighted in the assembly component.
Register	Displays disassembled instructions starting at the address stored in the source register. The instruction starting at the address stored in the register is highlighted.

2.6.3 Dragging from the Data Component

Table 2-3 shows what occurs when dragging from the data component.

Table 2-3. Dragging from the Data Component

Destination Component	Operation
Command Line	Dragging the name appends the address range of the variable to the current command in the command line window. Dragging the value appends the variable value to the current command in the command line window.
Memory	Dumps memory starting at the address where the selected variable is located. The memory area where the variable is located is selected in the memory component. Dragging from a data value in the memory component is not allowed.
Register	Dragging the name loads the destination register with the address of the selected variable. Dragging the value loads the destination register with the value of the variable.

NOTE: *The user can drag either a variable name or a variable value. Both operations are possible. Dragging the variable name drags the address of the variable.*

Expressions are evaluated at run time; therefore, they do not have a location address associated with them. Without a location address, the user cannot drag an expression name into another component, although expression values can be dragged to other components.

2.6.4 Dragging into the Data Component

Table 2-4 shows all options available when dragging into the data component.

Table 2-4. Dragging into the Data Component

Source Component	Operation
Source	A selection in the source window is considered as an expression in the data window, as if it had been entered through the expression editor of the data component (refer to ***3.3.6.1 Expression Editor***).
Module	Displays global variables from the selected module in the data component

2.6.5 Dragging from the Source Component

Table 2-5 describes the actions taken when dragging from the source component.

Table 2-5. Dragging from the Source Component

Destination Component	Operation
Assembly	Displays disassembled instructions starting at the first high level language instruction selected. The assembler instructions corresponding to the selected high level language instructions are highlighted in the assembly component.
Register	Loads the destination register with the PC of the first instruction selected
Data	A selection in the source window is considered as an expression in the data window, as if it had been entered through the expression editor of the data component (refer to ***3.3.6.1 Expression Editor***).

2.6.6 Dragging into the Source Component

Table 2-6 decribes the action taken when dragging into the source component.

Table 2-6. Dragging into the Source Component

Source Component	Operation
Assembly	Source component scrolls to the source statements corresponding to the pointed to assembly instruction and highlights it.
Memory	Displays high level language source code starting at the first address selected. The instructions corresponding to the selected memory area are greyed in the source component.
Module	Displays source code from the selected module

2.6.7 Dragging from the Memory Component

Table 2-7 describes the action taken when dragging from the memory component.

Table 2-7. Dragging from the Memory Component

Destination Component	Operation
Assembly	Displays disassembled instructions starting at the first address selected. The instructions corresponding to the selected memory area are highlighted in the assembly component.
Command Line	Appends the selected memory range to the command line window
Register	Loads the destination register with the start address of the selected memory block
Source	Displays high level language source code starting at the first address selected. Instructions corresponding to the selected memory area are greyed in the source component.

2.6.8 Dragging into the Memory Component

Table 2-8 describes the action taken when dragging into the memory component.

Table 2-8. Dragging into the Memory Component

Source Component	Operation
Assembly	Dumps memory starting at the selected instruction PC. The PC location is selected in the memory component.
Data	Dumps memory starting at the address of the selected variable. The memory area where the variable is located is selected in the memory component.
Register	Dumps memory starting at the address stored in the selected register. The corresponding address is selected in the memory component.
Module	Dumps memory starting at the address of the first global variable in the module. The memory area where this variable is located is selected in the memory component.

2.6.9 Dragging from the Register Component

The options shown in Table 2-9 are available when dragging from the register component.

Table 2-9. Dragging from the Register Component

Destination Component	Operation
Assembly	Assembly component receives an address range, scrolls to the corresponding instruction, and highlights it.
Memory	Dumps memory starting at the address stored in the selected register. The corresponding address is selected in the memory component.
Command Line	The address stored in the pointed to register is appended to the current command.

2.6.10 Dragging into the Register Component

The options shown in Table 2-10 are available when dragging into the register component.

Table 2-10. Dragging into the Register Component

Source Component	Operation
Assembler	Loads the destination register with the address of the selected instruction
Data	Dragging the name loads the destination register with the start address of the selected variable. Dragging the value loads the destination register with the value of the variable.
Source	Loads the destination register with the PC of the first instruction selected
Memory	Loads the destination register with the start address of the selected memory block

2.6.11 Dragging from the Module Component

The options shown in Table 2-11 are available when dragging from the module component.

Table 2-11. Dragging from the Module Component

Destination Component	Operation
Data | Global	Displays the global variables from the selected module in the data component
Memory	Dumps memory starting at the address of the first global variable in the module. The memory area where this variable is located is selected in the memory component.
Source	Displays source code from the selected module

NOTE: *Nothing can be dragged into the module component.*

2.7 MCUez Debugger Main Menu

The main window acts as a container for the other component windows. Additionally, it provides a global menu bar, toolbar, status bar, and an information bar for individual component windows. Figure 2-5 shows the debugger **Main** menu.

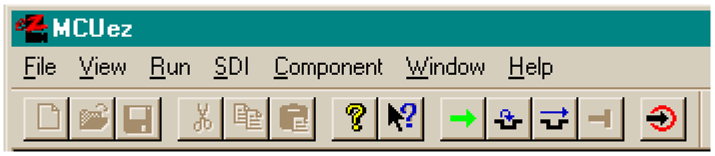


Figure 2-5. Debugger Main Menu

2.7.1 File Menu

The **File** menu is dedicated to the debugger. Use the **File** menu to exit the debugger, as shown in Figure 2-6.

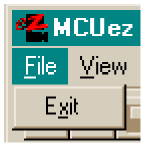


Figure 2-6. File Menu

2.7.2 View Menu

The **View** menu contains two options:

- Check / uncheck toolbar to display or hide it.
- Check / uncheck status bar to display or hide it.

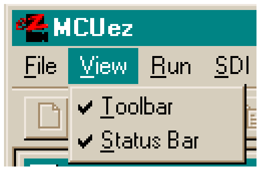


Figure 2-7. View Menu

2.7.3 Run Menu

The **Run** menu is used for debug operations. Table 2-12 defines all **Run** menu commands.

Table 2-12. Run Menu

Menu Entry	Description
Start/Continue	Starts or continues execution of the loaded application until a breakpoint is reached, a runtime error is detected, or the user stops the application using **Halt**.
Halt	Interrupts and halts a running application enabling examination of the state of each variable in the application, set breakpoints, and inspect the source code.
Assembly Step	If the application is halted, this command performs a single step; for example, execution continues for one CPU instruction from the point it was halted.
Assembly Step Over	Similar to the Assembly Step command, but does not step into called functions (steps over subroutine call instructions)
Breakpoints	Opens the **Breakpoints Setting** dialog, which lists breakpoints defined in the application and allows their properties to be modified. Breakpoints also can be set in the **Source** and **Assembly** component windows.

The menu entries (except breakpoints) have an associated toolbar button. Figure 2-8 illustrates the **Run** menu.

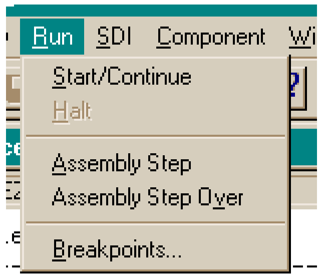
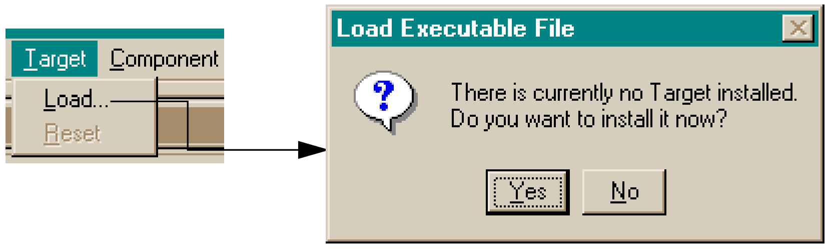


Figure 2-8. Run Menu

2.7.4 Target Menu

The **Target** menu appears between the **Run** and **Component** menus when no target is specified in the *project.ini* file and no target has been set. Select **Target | Load...** to display the Load Executable File message. When connected to a target, the **Target** menu is replaced by a new menu with the name of the target, for example D-Bug12.

Also select **Component | Set Target...** to connect to a target.



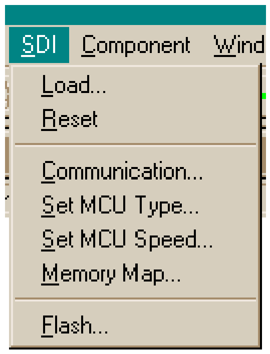
There is currently no Target installed.
Do you want to install it now?

Figure 2-9. Target Menu: Load and Reset

If the Load Executable File message is displayed, the target is not set and no application files (*.abs*) can be loaded. Click **Yes** to display the **Set Target** listbox, enabling selection of a target.

2.7.4.1 ESL Target

Initially, ESL (emulator server library) is the default target in the *project.ini* file. When communication is established between another target and the debugger, the **ESL Target** menu is replaced by the new target name, for example SDI (serial device interface). If the target is not identified, the **ESL** menu remains. **Figure 2-10** depicts the **SDI** and **ESL** menus.



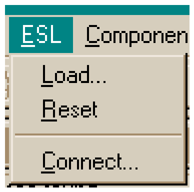


Figure 2-10. SDI and ESL Target Menus

2.7.5 Component Menu

Open additional component windows by selecting the **Component | Open...** menu entry. Select a window component from the list of components and click **OK**. See **Section 3. Component Windows**. The display font and background color for the debugger environment can also be changed. **Figure 2-11** illustrates the **Component** menu.

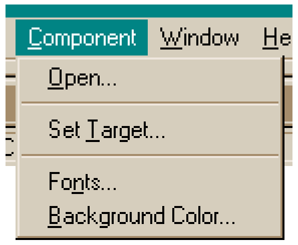


Figure 2-11. Component Menu

Select **Component | Set Target...** to set the preferred target.

Select **Component | Fonts...** to open a standard **Font Selection** dialog and select the font to use in the component windows.

Select **Component | Background Color...** to open the color selection dialog. Select a background color for component windows.

2.7.6 Window Menu

The **Window** menu sets the general arrangement of component windows and loads or stores arrangements.

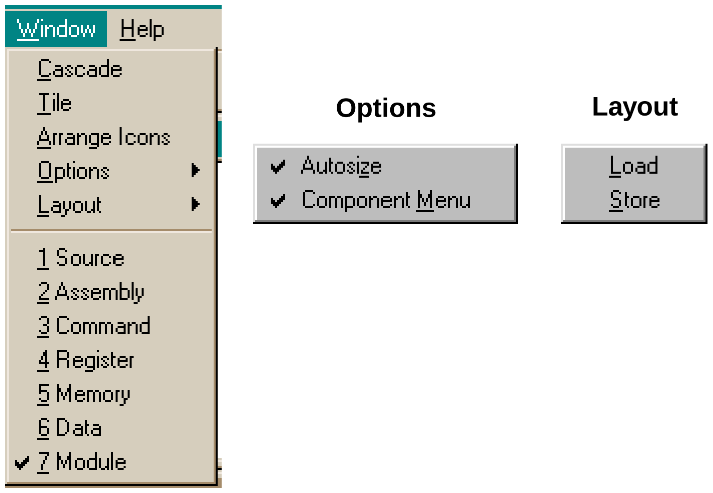


Figure 2-12. Window Menu

Check **Autosize** to automatically resize component windows when the debugger main window is resized. Check **Component Menu** to display the menu associated with the currently selected component window. For example, if the **Source** window is selected, the **Source** menu is displayed in the main menu.

Select **Window | Layout** to load or store arrangements in a *.hwl* file.

2.7.7 Help Menu

The **Help** menu provides on-line help and specific information about a topic.

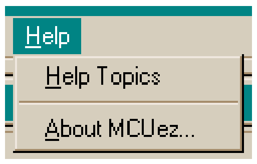


Figure 2-13. Help Menu

Section 3. Component Windows

3.1 Contents

3.2 Introduction

This section describes the various component windows and how to use them.

3.3 Components

The MCUez debugger operates in an environment constructed of windows that represent components of the target system. Component windows are applications loaded at run-time and have access to the target interface, the symbol table, and other global facilities. Additional component windows can be opened by selecting the **Component|Open...** menu option. **Figure 3-1** shows the **Open Window Component** dialog box. Component windows are implemented as dynamic link library files with the extension *.wnd*.

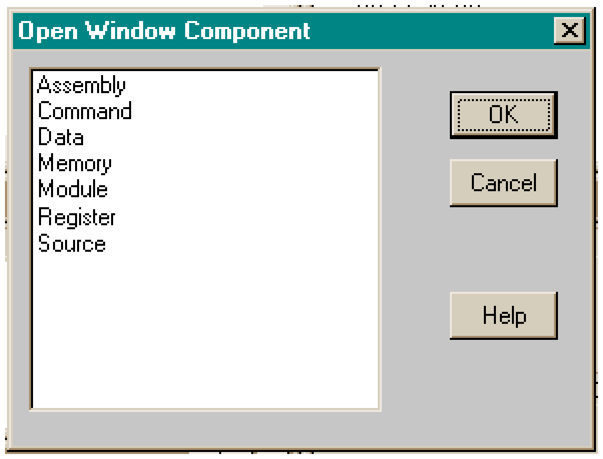


Figure 3-1. Open Window Component Dialog Box

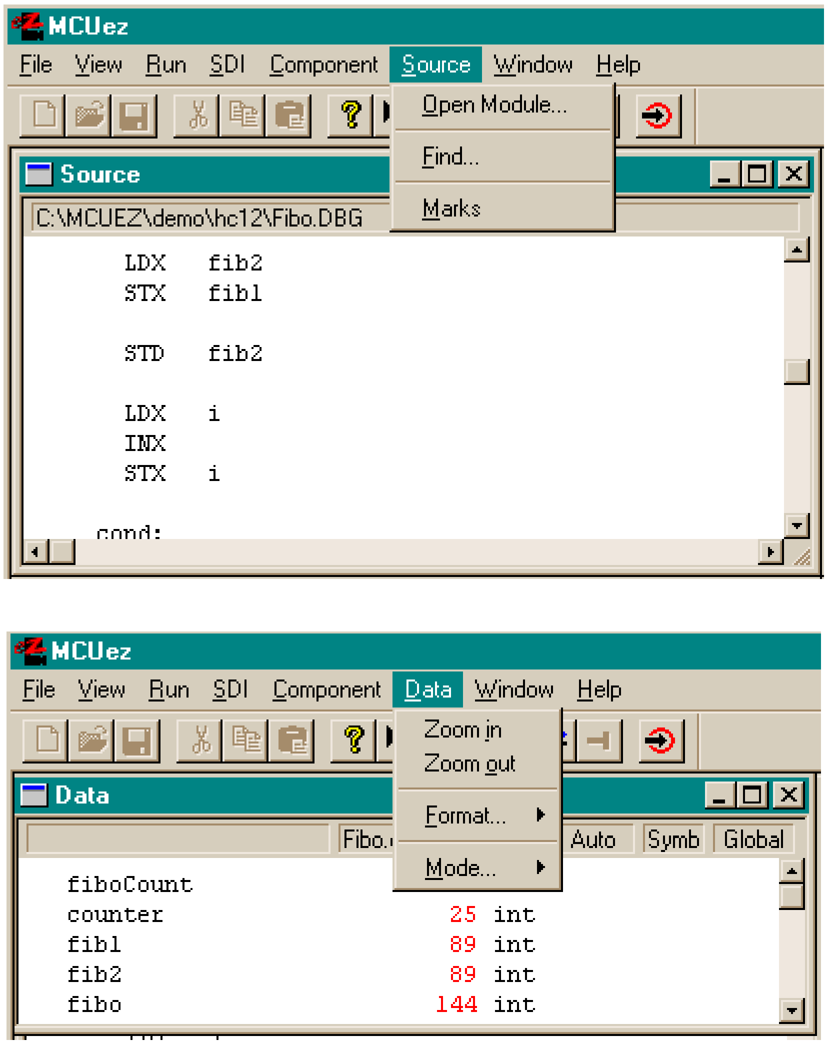
CPU (central processor unit) components handle processor-specific operations such as register naming, instruction decoding (disassembly), and stack tracing. CPU components are reflected in the **Register** and **Memory** component windows. Applicable CPU components are loaded when connection with the target is established.

The target is connected by an emulator, a ROM monitor, or any other supported connection device. Only one target can be loaded at any time, such as SDI or D-Bug12.

3.3.1 Component Window Menus

Each component window has two menus. One menu is on the main menu bar and the other is a pop-up menu that is opened by clicking the right mouse button in the active component window.

On the main menu bar, the active component menu is located between the **Component** menu and **Window** menu. For example, if the **Source** component window is activated, the **Source** menu will appear. If the **Data** component window is activated, the **Data** menu will appear as shown in **Figure 3-2**.



C:\MCUEZ\demo\hc12\Fibo.DBG

```
        LDX    fib2
        STX    fib1

        STD    fib2

        LDX    i
        INX
        STX    i

    cond:
```

```
   fiboCount
   counter                25  int
   fib1                   89  int
   fib2                   89  int
   fibo                  144  int
```

Figure 3-2. Active Component Menu

The pop-up menu is a dynamic context-sensitive menu. It contains entries for the currently active component window. Pop-up menu entries differ to reflect operations that can be performed on the object pointed to by the mouse. For example, if a breakpoint is pointed to, additional options are available to delete or disable the breakpoint. **Figure 3-3** shows the **Source** component pop-up menus.

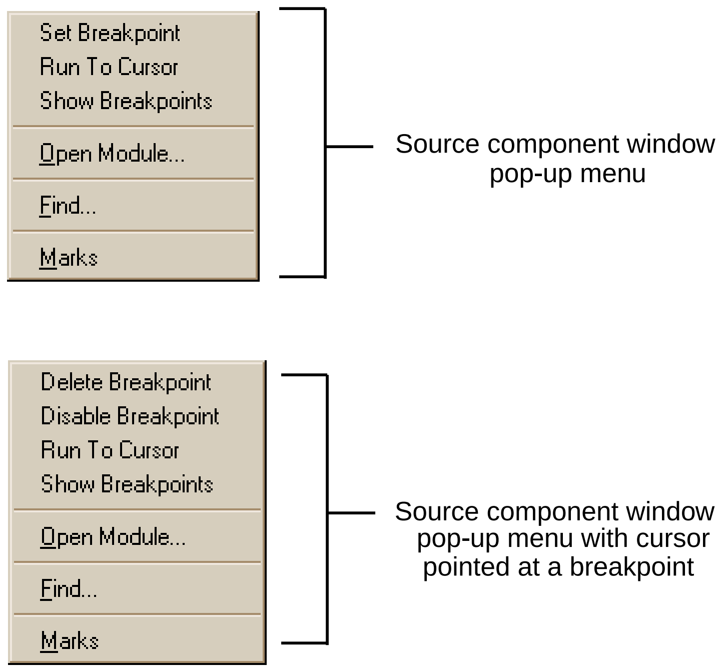


Figure 3-3. Source Component Pop-up Menus

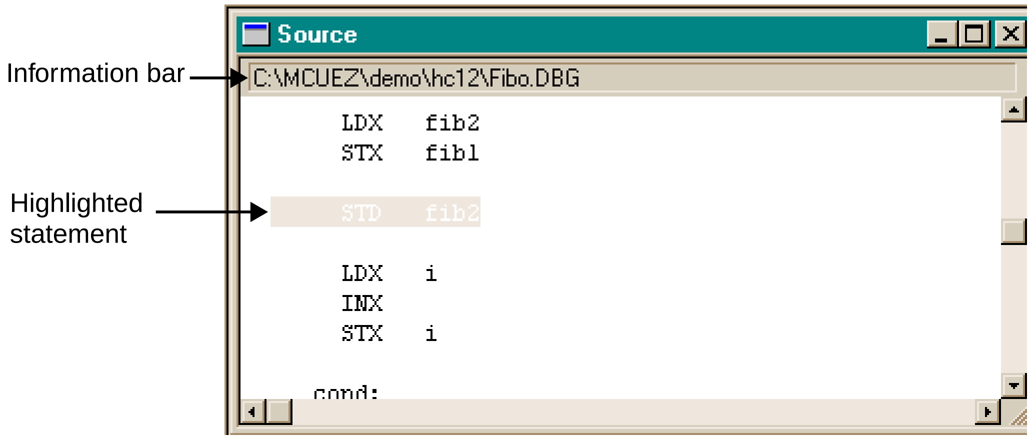
3.3.2 Source Component Window

The **Source** component window displays the program source code (application file). It enables the user to view, change, monitor, and control the current execution location in the program. A word is selected by placing the mouse pointer on the word, then double clicking the left mouse button. A section of code is selected by holding down the left mouse button and dragging the mouse across the code to highlight the selected area.

NOTE: *Text displayed in the* **Source** *component window cannot be edited. The* **Source** *component window is a file viewer only.*

If breakpoints have been set in the program, they will be marked with a special symbol to indicate the nature of the breakpoint. (See **3.3.2.1 Breakpoints**.)

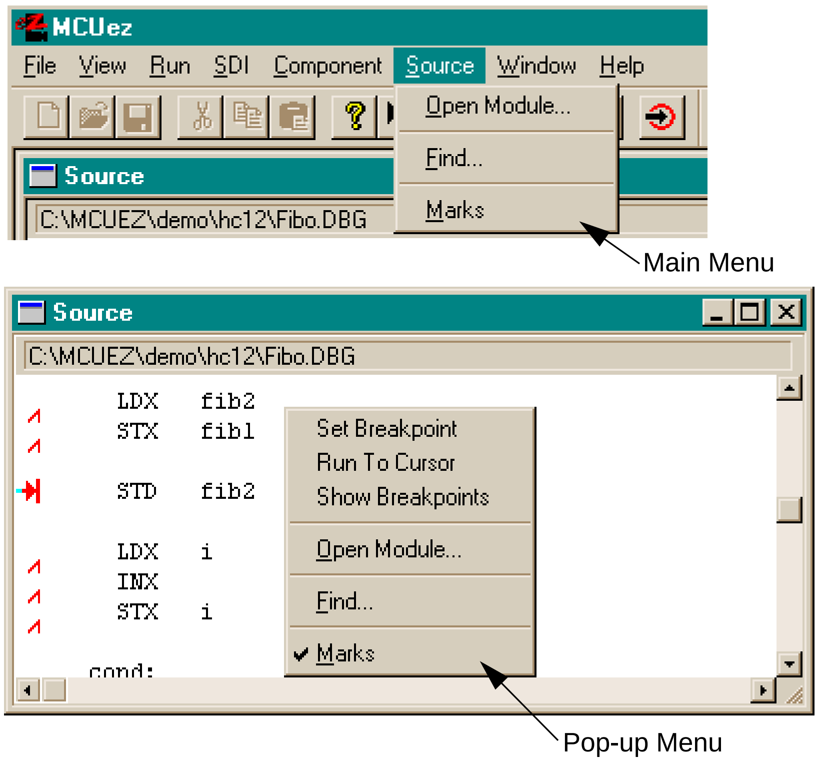
If execution has stopped, the current position is marked in the **Source** component window by highlighting the corresponding statement. The complete path of the displayed source file is displayed in the information bar. **Figure 3-4** shows the **Source** component window.



```
C:\MCUEZ\demo\hc12\Fibo.DBG
        LDX    fib2
        STX    fib1

        STD    fib2

        LDX    i
        INX
        STX    i

      cond:
```

Figure 3-4. Source Component Window

Figure 3-5 shows the **Source** component menu and its associated pop-up menu obtained by clicking the right mouse button.



```
Source
C:\MCUEZ\demo\hc12\Fibo.DBG
        LDX     fib2
        STX     fib1

        STD     fib2

        LDX     i
        INX
        STX     i

      cond:
```

Figure 3-5. Source Component Main Menu and Pop-up Menu

The following describes each **Source** component menu entry.

- Main menu
  - **Open Module** — Opens a dialog which lists all source files bound to the application currently loaded. Double click on the module to be viewed in the source window.
  - **Find** — Opens a dialog box prompting for a string and then searches the file displayed in the source component for the specified string

    To start searching, click **Find Next**. The search begins at the current selection or first line visible in the source component. **Up/down** radio buttons enable the search backward or forward. If the string is found, the source component selection is positioned at the string. If the string is not found, a message will be displayed.
  - **Marks** — Toggles the display of source positions where breakpoints may be set. If this menu entry is checked, the positions where a breakpoint can be set are marked by small triangles.

- Pop-up menu
  - **Set Breakpoint** — Appears in the **Source** pop-up menu if no breakpoint is set or disabled at the nearest code position (visible with marks). When selected, it sets a permanent breakpoint at this position. If program execution reaches this statement, the program is halted and the current program state is displayed in all window components.
  - **Delete Breakpoint** — Appears in the pop-up menu if a breakpoint is set or disabled at the nearest code position (visible with marks). When selected, it deletes the breakpoint.
  - **Enable Breakpoint** — Appears if a breakpoint is disabled at the nearest code position (visible with marks). When selected, enables the breakpoint.
  - **Disable Breakpoint** — Appears if a breakpoint is set at the nearest code position (visible with marks). When selected, it disables the breakpoint.
  - **Run to Cursor** — When selected, sets a temporary breakpoint at the nearest code position and continues program execution. When program execution reaches this instruction, the program is halted and the current program state is displayed in all window components.

    If there is a disabled breakpoint at this position, the temporary breakpoint will be disabled also and the program will not halt. Temporary breakpoints are automatically removed as soon as they are reached.
  - **Show Breakpoints** — Opens the **Breakpoints Setting** dialog box and allows viewing of the list of breakpoints defined in the application and modification of their properties.
  - **Open Module** — Opens a dialog which lists all source files bound to the application currently loaded. Double click on the module to be viewed in the source window.
  - **Find** — Opens a dialog box prompting for a string and then searches the file displayed in the source component for the specified string
  - **Marks** — Toggles the display of source positions where breakpoints may be set. If checked, positions where a breakpoint can be set are marked by upside down check marks.

3.3.2.1 Breakpoints

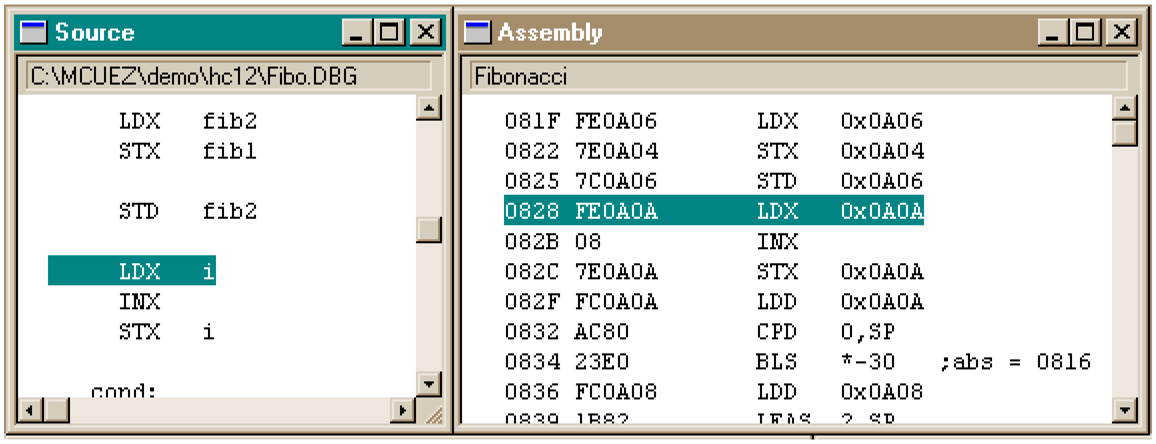
If breakpoints have been set in the program, they will be marked with a special symbol, depending on the kind of breakpoint.

- A temporary breakpoint has this symbol: A lighter color encased by a darker color with the lighter color usually being yellow and the darker being red.
- A permanent breakpoint has this symbol: The color is solid red.
- A disabled breakpoint looks like: The breakpoint is half normal density and a light red color.

If execution has stopped, the current position is marked in the source component by highlighting the corresponding statement. The path of the displayed source file is shown in the information bar.

3.3.2.2 Decoding Instructions

To disassemble code, select a range of instructions in the source component and drag it into the assembly component. The corresponding range of code is highlighted in the **Assembly** component window. See **Figure 3-6**.



```
      LDX    fib2
      STX    fib1

      STD    fib2

      LDX    i
      INX
      STX    i

    cond:
```

```
081F  FE0A06        LDX    0x0A06
0822  7E0A04        STX    0x0A04
0825  7C0A06        STD    0x0A06
0828  FE0A0A        LDX    0x0A0A
082B  08            INX
082C  7E0A0A        STX    0x0A0A
082F  FC0A0A        LDD    0x0A0A
0832  AC80          CPD    0,SP
0834  23E0          BLS    *-30    ;abs = 0816
0836  FC0A08        LDD    0x0A08
```

Figure 3-6. Online Disassembly

3.3.2.3 Find Dialog Box

Enter the string to search for in the **Find what** edit box. To start searching, click **Find Next**. The search begins at the current selection or first line visible in the source component when nothing is selected.

Figure 3-7 shows the **Find** dialog box.

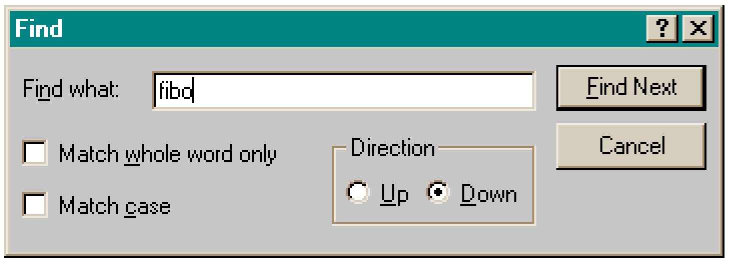


Figure 3-7. Find Dialog Box

The dialog box enables the following options to be specified:

- **Match whole word only**: If this box is checked, only strings that are separated by special characters are recognized.
- **Match case**: If this box is checked, the search is case sensitive.
- The **Up / Down** buttons will enable the search backward or forward. If the string is found, the source component selection is positioned at the string. If the string is not found, a message is displayed.

NOTE: *If an item (single word or source section) has been selected in the source component before opening the* **Find** *dialog, the first line of the selection will be automatically copied into the* **Find what** *edit box.*

3.3.3 Assembly Component Window

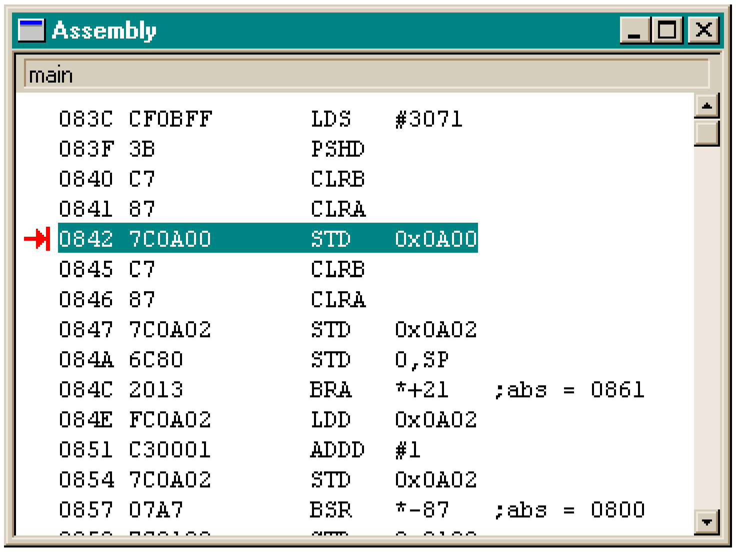
The assembly component displays program code in disassembled form. The assembly component function is similar to that of the source component, enabling the user to view, change, monitor, and control the current location of program execution.

The assembly component contains all the on-line disassembled instructions generated by the application. Each disassembled line shows this information:

- Address
- Machine code
- Instruction
- Absolute address in case of a branch instruction

Per default, the instruction and absolute address for a branch instruction are visible. The program instruction and absolute address also can be viewed by selecting the corresponding menu entry.

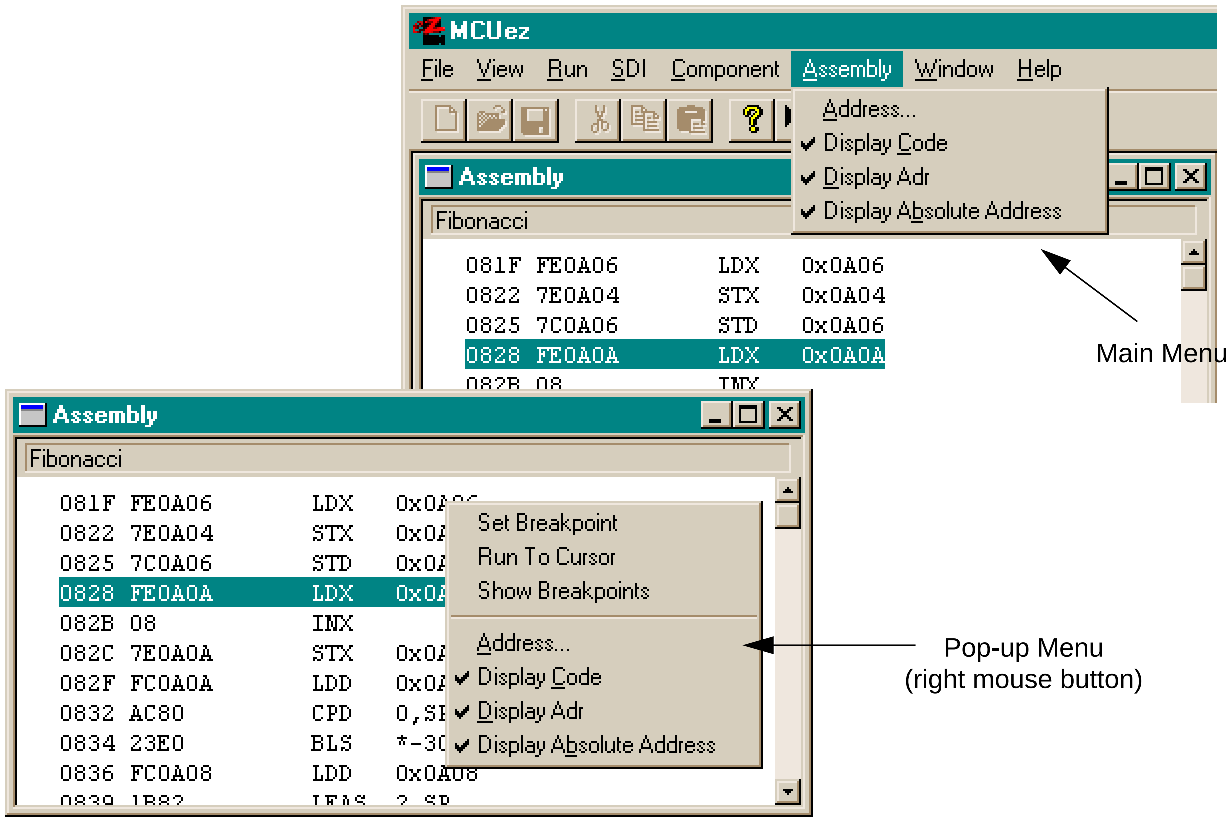
If breakpoints have been set in the application, they are marked in the assembly component with a special symbol, depending on the type of breakpoint. If execution has stopped, the current position is marked in the assembly component by highlighting the corresponding instruction. The information bar displays the procedure name that contains the currently selected instruction. **Figure 3-8** shows the **Assembly** component window.



```
083C  CF0BFF        LDS   #3071
083F  3B            PSHD
0840  C7            CLRB
0841  87            CLRA
0842  7C0A00        STD   0x0A00
0845  C7            CLRB
0846  87            CLRA
0847  7C0A02        STD   0x0A02
084A  6C80          STD   0,SP
084C  2013          BRA   *+21    ;abs = 0861
084E  FC0A02        LDD   0x0A02
0851  C30001        ADDD  #1
0854  7C0A02        STD   0x0A02
0857  07A7          BSR   *-87    ;abs = 0800
```

Figure 3-8. Assembly Component Window

Figure 3-9 shows the **Assembly** component main menu and associated pop-up menu. The bulleted list that follows describes each menu selection.



```
081F  FE0A06        LDX   0x0A06
0822  7E0A04        STX   0x0A04
0825  7C0A06        STD   0x0A06
0828  FE0A0A        LDX   0x0A0A
082B  08            INX
```

```
081F  FE0A06        LDX   0x0A06
0822  7E0A04        STX   0x0A
0825  7C0A06        STD   0x0A
0828  FE0A0A        LDX   0x0A
082B  08            INX
082C  7E0A0A        STX   0x0A
082F  FC0A0A        LDD   0x0A
0832  AC80          CPD   0,SP
0834  23E0          BLS   *-30
0836  FC0A08        LDD   0x0A08
0839  1B82          LEAS  2,SP
```

Figure 3-9. Assembly Component Main Menu and Associated Pop-up Menu

- Main menu
  - **Address...** — Opens a dialog box prompting for an address. If a hexadecimal address is entered, memory contents are interpreted and displayed as assembler instructions starting at the specified address.
  - **Display Code** — Displays the machine code in front of each disassembled instruction
  - **Display Address** — Displays the location address at the beginning of each disassembled instruction. If both **Display Code** and **Display Address** are selected at the same time, the absolute address is displayed first, then the hexadecimal code, and finally the disassembled instruction.
  - **Display Absolute Address** — Displays the absolute address at the end of the disassembled instruction for a branch instruction

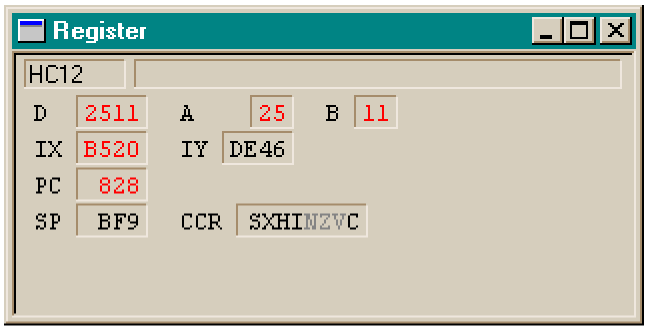
- Pop-up menu
  - **Set Breakpoint** — Appears only in the pop-up menu if no breakpoint is set or disabled on the pointed to instruction. When selected, it sets a permanent breakpoint on this instruction. When program execution reaches this instruction, the program is halted and the current program state is displayed in all window components.
  - **Delete Breakpoint** — Appears only in the pop-up menu if a breakpoint is set or disabled on the pointed to instruction. When selected, deletes this breakpoint.
  - **Enable Breakpoint** — Appears in the pop-up menu if a breakpoint is disabled on the pointed to instruction. When selected, it enables this breakpoint.
  - **Disable Breakpoint** — Appears in the pop-up menu if a breakpoint is set on the pointed to instruction. When selected, it disables this breakpoint.
  - **Run to Cursor** — When selected, sets a temporary breakpoint on the pointed to instruction and continues program execution. When program execution reaches this instruction, the program is halted and the current program state is displayed in all window components. If there is a disabled breakpoint at this position, the temporary breakpoint will also be disabled and the program will not halt. Temporary breakpoints are removed automatically as soon as they are reached.
  - **Show Breakpoints** — Opens the **Breakpoints Setting** dialog box and lists breakpoints defined in the application. Breakpoint properties can then be modified.
  - **Address** — Opens a dialog box prompting for an address. If a hexadecimal address is entered, memory contents are interpreted and displayed as assembler instructions starting at the specified address.
  - **Display Code** — Displays the machine code in front of each disassembled instruction
  - **Display Address** — Displays the location address at the beginning of each disassembled instruction. If both **Display Code** and **Display Address** are selected, the absolute address is displayed first, then the hexadecimal code, and finally the disassembled instruction.
  - **Display Absolute Address** — For a branch instruction, displays the absolute address at the end of the disassembled instruction

3.3.3.1 Retrieving Source Statements

Point to an instruction in the **Assembly** component window and drag and drop it into the **Source** component window. The source component scrolls to the source statement that generates this assembly instruction and highlights it.

3.3.4 Register Component Window

The **Register** component window displays the content of registers and status register bits of the target processor. The **Register** window changes to reflect the target processor being accessed. Register values are displayed in binary or hexadecimal format. The system allows editing of all values. **Figure 3-10** shows the **Register** component window.



```
D   2511    A    25   B 11
IX  B520    IY DE46
PC   828
SP   BF9    CCR SXHINZVC
```

Figure 3-10. Register Component Window

3.3.4.1 Status Register Bits

All status register bits that are set, are displayed dark. All reset status register bits are displayed gray. A bit is toggled by placing the mouse pointer on the bit, then double clicking the left mouse button. Contents of registers that have changed since the last display refresh are shown in red (except for status register bits) during application execution.

3.3.4.2 Editing Registers

Double clicking on a register opens an edit box over the register enabling modification of the register value.

The modified value is not validated if the Escape key is pressed. If Esc is pressed, the content of the register remains unchanged. If the Enter key is pressed or if selected outside the edited register, the input value is validated and the register content is changed.

If the Tab key is pressed, the register content is changed and validated, and the next register value is selected for modification. Double clicking a bit in the status register toggles the selected bit.

Click and hold the left mouse button and press the A key to view the source code as well as the changed contents of the assembly and memory components. The source component shows the source code located at the address stored in the register. The assembly component shows the disassembled code starting at the address stored in the register. The memory component dumps memory starting at the address stored in the register. Clicking the right mouse button opens the register component pop-up menu.

3.3.4.3 Register Display Options Menu

The **Register Display Options** menu (**Figure 3-11**) provides the option to display code in either binary or hexadecimal format.

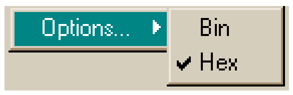


Figure 3-11. Register Display Options Menu

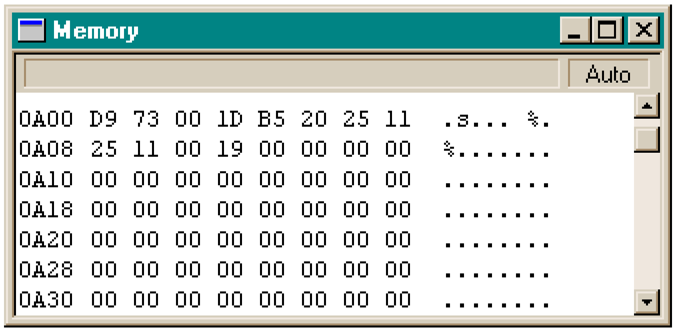
3.3.5 Memory Component Window

The memory component displays unstructured memory contents or memory dumps (continuous memory words without distinction between variables). Various word sizes (byte, word, double) and data formats (binary, octal, hexadecimal, decimal, unsigned decimal) can be specified for memory display.

To specify the start address for the memory dump, use the **Address** menu entry. A memory area can be initialized with a fill pattern using the **Fill** dialog box. An ASCII dump can be added/removed on the right side of the numerical dump when checking/unchecking ASCII in the **Display** menu entry. The location address also can be added/removed on the left side of the numerical dump when checking/unchecking **Address** in the **Display** menu entry.

NOTE: *Memory values that have changed since the previous program halt are displayed in red. If a memory item is edited or rewritten with the same value, the memory item display remains black.*

The object information bar contains the procedure or variable name, structure field, and memory range matching the first selected memory word. Figure 3-12 shows the **Memory** component window.



```
0A00  D9  73  00  1D  B5  20  25  11   .s... %.
0A08  25  11  00  19  00  00  00  00   %.......
0A10  00  00  00  00  00  00  00  00   ........
0A18  00  00  00  00  00  00  00  00   ........
0A20  00  00  00  00  00  00  00  00   ........
0A28  00  00  00  00  00  00  00  00   ........
0A30  00  00  00  00  00  00  00  00   ........
```

Figure 3-12. Memory Component Window

3.3.5.1 Memory Component Operations

Memory component operations are:

- Double click a memory position to edit it.
- Drag the mouse in the memory dump to select a memory range.
- Press and hold the left mouse button and press the A key to jump to a memory address. The selected value is interpreted as an address. The memory component dumps memory starting at this address.

3.3.5.2 Memory Component Pop-up Menu

The **Memory** component pop-up menu (**Figure 3-13**) is displayed by placing the cursor in the active (selected) **Memory** component window and clicking the right mouse button.

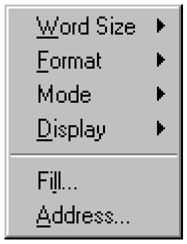


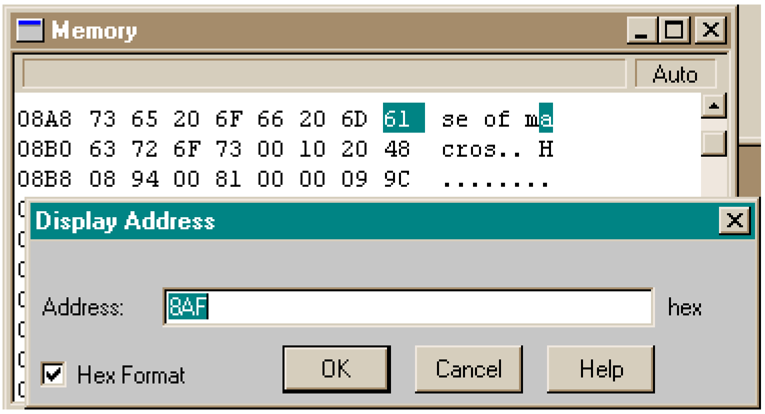
Figure 3-13. Memory Component Pop-up Menu

Table 3-1 defines the entries in the **Memory** component pop-up menu.

Table 3-1. Memory Component Pop-up Menu

Menu Entry	Description
Word Size	Opens a submenu enabling the user to specify the display unit size. The three available sizes are byte, word (= 2 bytes), and longword (= 4 bytes).
Format	Selects the format in which the items are to be displayed. Available formats are hexadecimal, binary, octal, signed, and unsigned decimal.
Display	Opens a submenu enabling the user to toggle the display of addresses and ASCII dump
Mode	Switches between automatic, periodical, and frozen update modes (See Section **3.3.5.3 Memory Update Mode**.)
Address	Opens the memory dialog and prompts for an address. The memory component dumps memory starting at the specified address.
Fill	Opens the **Fill** dialog to fill a memory range with a bit pattern

Address ... opens a dialog as shown in **Figure 3-14**.



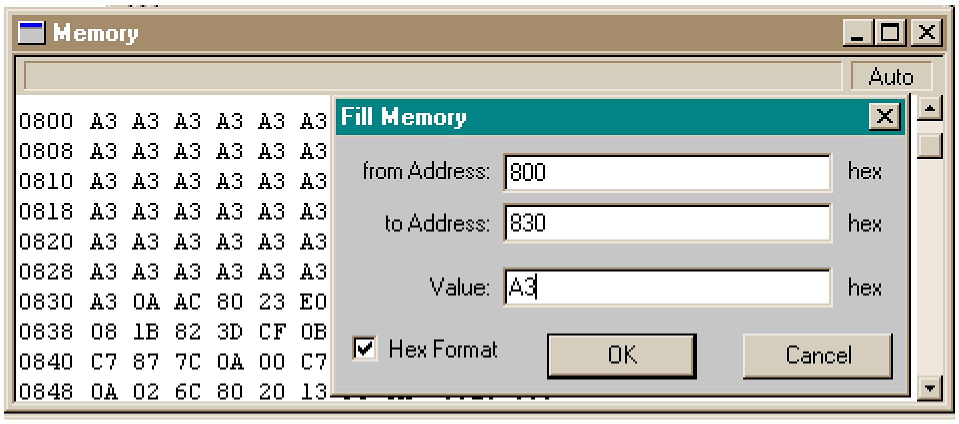
```
08A8  73 65 20 6F 66 20 6D 61  se of ma
08B0  63 72 6F 73 00 10 20 48  cros.. H
08B8  08 94 00 81 00 00 09 9C  ........
```

Address: 8AF

☑ Hex Format

Figure 3-14. Memory Component Display Address

The **Fill** menu entry opens a dialog (**Figure 3-15**) to fill a memory range with a bit pattern.



```
0800  A3  A3  A3  A3  A3  A3
0808  A3  A3  A3  A3  A3  A3
0810  A3  A3  A3  A3  A3  A3
0818  A3  A3  A3  A3  A3  A3
0820  A3  A3  A3  A3  A3  A3
0828  A3  A3  A3  A3  A3  A3
0830  A3  0A  AC  80  23  E0
0838  08  1B  82  3D  CF  0B
0840  C7  87  7C  0A  00  C7
0848  0A  02  6C  80  20  13
```

from Address: 800

to Address: 830

Value: A3

☑ Hex Format

Figure 3-15. Memory Component Fill Memory Dialog Box

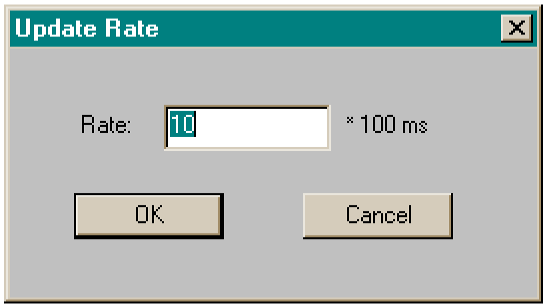
Click **OK** in the **Fill Memory** dialog box to initialize all memory positions from $800 to $830 with the value $A3.

NOTE: *If* **Hex Format** *is checked, numbers and letters are considered to be hexadecimal numbers. Otherwise, expressions can be typed and hex numbers must be prefixed with Ox or $.*

3.3.5.3 Memory Update Mode

The memory component can be updated in three different modes:

1. In automatic mode (default), memory dump is updated when the target is stopped.
2. In frozen mode, memory dump displayed in the memory component is not updated when the target is stopped.
3. In periodical mode, memory dump is updated at regular time intervals when the application is running. The default update rate is 1 second, but it can be modified by steps up to 100 ms using the associated dialog box.



Rate: 10 * 100 ms

Figure 3-16. Update Rate: Memory Component

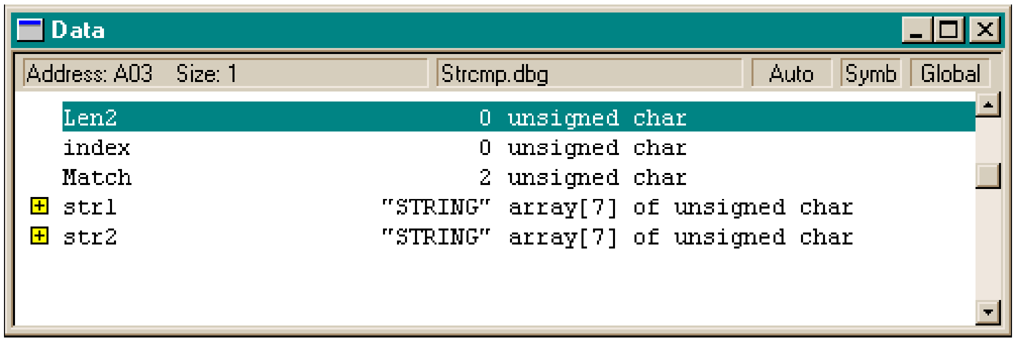
NOTE: *The periodic mode is not available for all targets. Additional configurations may be required to make it work. Refer to the specific target manual.*

3.3.6 Data Component Window

The data component contains the names, values, and types of variables. The **Data** component window shows all variables present in the current source module. Display formats, such as symbolic representation, (depending on the variable types), as well as hexadecimal, octal, binary, signed, and unsigned formats are selectable.

The information bar contains the address and size of the selected variable. It also contains the module name where the displayed variables are defined, the display mode (automatic, frozen, etc.), and the display format (symbolic, hex, bin, etc.).

Values can be edited by double clicking on a value or the line containing a value. Arrays can be expanded by clicking on the plus (+) symbol preceding an array name. Figure 3-17 shows the **Data** component window.

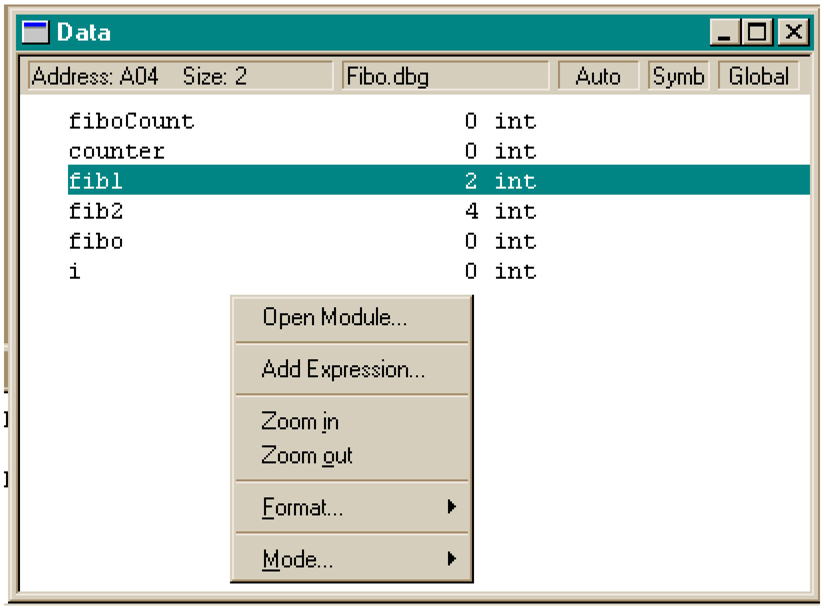


```
Len2                     0  unsigned char
index                    0  unsigned char
Match                    2  unsigned char
⊞ str1            "STRING"  array[7] of unsigned char
⊞ str2            "STRING"  array[7] of unsigned char
```

Figure 3-17. Data Component Window

3.3.6.1 Expression Editor

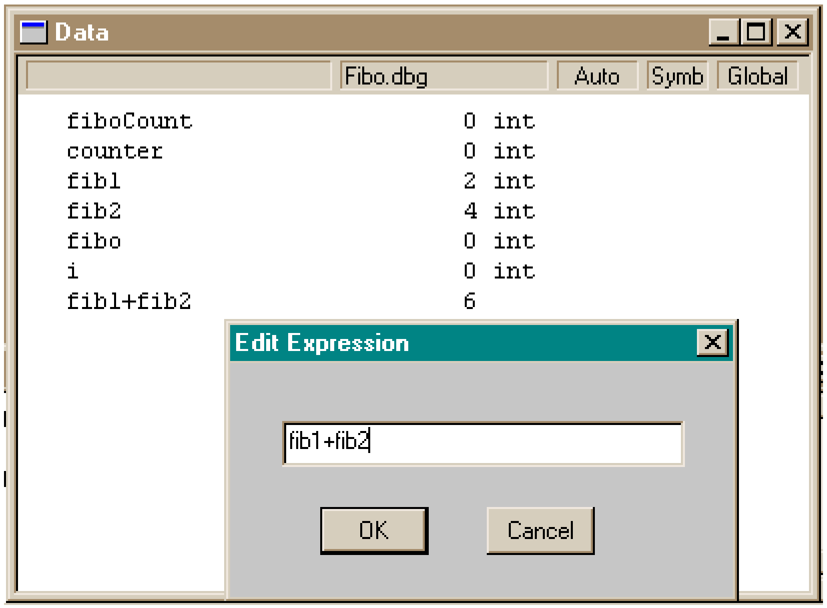
To add an expression, double click on a blank line in the data component to open the **Expression Editor** dialog or point to a blank line and right click to select **Add Expression...** in the pop-up menu (Figure 3-18).



```
fiboCount     0  int
counter       0  int
fib1          2  int
fib2          4  int
fibo          0  int
i             0  int
```

Figure 3-18. Accessing the Expression Editor

Enter a logical or numerical expression in the **Edit Expression** box using ANSI C syntax. This expression is a function of one or several variables from the current data component. Figure 3-19 shows the expression editor.



```
fiboCount        0  int
counter          0  int
fib1             2  int
fib2             4  int
fibo             0  int
i                0  int
fib1+fib2        6
```

fib1+fib2

Figure 3-19. Using the Edit Expression Box

Example:

With two variables `variable_1, variable_2;`

- Expression entered:
  `(variable_1<<variable_2)+ 0xFF <= 0x1000`
  will result as a Boolean type
- Expression entered:
  `(variable_1>>~variable_2)* 0x1000`
  will result as an integer type

NOTE: *It is not possible to drag an expression defined with the expression editor.*

3.3.6.2 Data Component Pop-up Menus

Figure 3-20 shows the **Data** component pop-up menus and **Table 3-2** identifies data component operations.

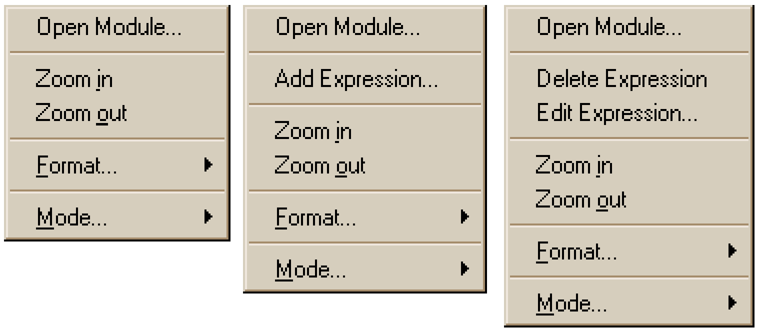


Figure 3-20. Data Component Pop-up Menus

Table 3-2. Data Component Operations

Menu Entry	Description
Open Module...	Opens a dialog that lists all source files bound to the application. The global variables from the selected module are displayed in the data component. This is only supported when the component is in global scope mode.
Zoom In	Expands the selected structure. For example, members of an array are displayed when selecting an array name and zooming in.
Zoom Out	Returns to the previous level
Format...	Switches between Symbolic (display format depends on the variable type), Hex (hexadecimal), Oct (octal), Bin (binary), Dec (signed decimal), UDec (unsigned decimal) display formats
Mode...	Switches between automatic, periodical, locked, and frozen update modes
Add Expression...	Appears only in the data pop-up menu when right clicking on an empty line. When selected, a user-defined expression in the data component can be added through the **Edit Expression** dialog.
Edit Expression...	Appears only in the data pop-up menu when right clicking on a line containing a user-defined expression. When selected, allows the user to edit the pointed to user-defined expression through the **Edit Expression** dialog.
Delete Expression	Appears only in the pop-up menu when right clicking on a line containing a user-defined expression. When selected, it deletes the pointed to user-defined expression.

3.3.6.3 Data Update Mode

The data component can be updated in three different modes:

1. In automatic mode (default), variables are updated when the target is stopped. Variables from the currently executed module are displayed in the data component.
2. In locked and frozen mode, variables from a specific module are displayed in the data component. In locked mode, data component variable values are updated when the target is stopped. In frozen mode, variables are not updated when the target is stopped.
3. In periodical mode, variables are updated at regular time intervals when the target is running. The default update rate is 1 second. The update rate can be modified in steps of up to 100 ms.

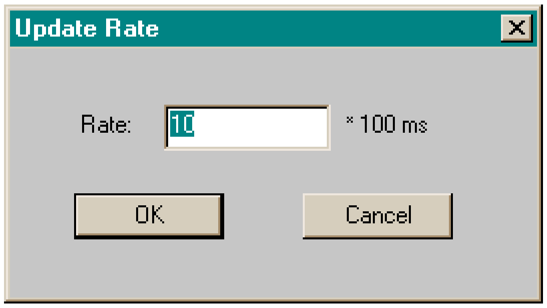


Figure 3-21. Update Rate: Data Component

3.3.7 Command Line Component Window

MCUez commands are entered on the right side of the `in>` terminal prompt in the **Command Line** component window. To recall a command (as in the DOS window), use the up arrow key, down arrow key, or special function key F3. The component executes the command entered and displays results or error messages. The 10 previous commands can be recalled using the up or down arrow keys. Commands are displayed in blue. Prompts and command responses are displayed in black. Error messages are displayed in red.

NOTE: *Memory ranges, addresses, and values can be dropped into the command line component. Nothing can be dragged from the command line component. The command line component accesses corresponding items of the current command. Refer to **Section 5. Command Reference** for more detailed information regarding MCUez commands.*

3.3.7.1 Command Line Component Operations

Figure 3-22 shows the **Command Line** component menu and **Table 3-3** describes the menu functions.

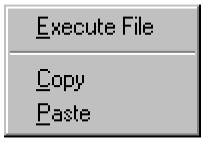


Figure 3-22. Command Line Component Menu

Table 3-3. Command Line Operations

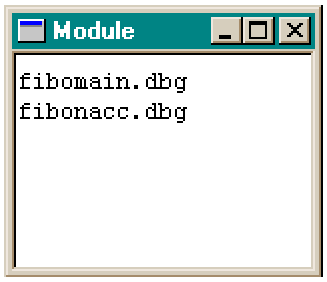
Menu Entry	Description
Execute File	When selected, opens a dialog where the user can select a file containing MCUez commands to be executed
Copy	Appears only in the pop-up menu when something is selected in the command line component. When selected, copies the selected text to the clipboard.
Paste	Appears only in the pop-up menu when something is stored in the clipboard. When selected, the first line of text currently stored in the clipboard is appended to the current command.

Selected text from the command line component can also be copied to the clipboard using the standard window key combination CTRL + C.

The first line of text currently stored in the clipboard can be appended to the current command using the standard window key combination CTRL + V.

3.3.8 Module Component Window

The **Module** component window lists source modules used to build the application. It displays all source files (source modules) bound to the application. The module component displays all modules in the order they appear in the absolute file (*.abs*). **Figure 3-23** shows the module component.



```
fibomain.dbg
fibonacc.dbg
```

Figure 3-23. Module Component Window

Double clicking a module name forces all open windows to display information about the module. The source component shows the module's source and the data component displays the module's global variables.

NOTE: *The module component has no associated menu.*

Section 4. Operating Procedures

4.1 Contents

4.2 Introduction

The MCUez debugger can be started from the **MCUez Shell** or an external editor. This chapter provides procedures on:

- Configuring the debugger
- Configuring the debugger for use with various editors
- Starting the debugger from editors, desktop, and shell
- Configuring a default layout
- Loading, starting, and stopping an application
- Defining and setting breakpoints
- Stepping through code within an application
- Working with variables
- Working on registers
- Accessing memory contents

4.3 Configuring the MCUez Debugger

The debugger must be associated with a project directory to find all requested configuration and component files.

The project (working) directory is defined in the *mcutools.ini* file located in the Windows® directory (for example `C:\winnt`). The working directory (including path) is defined in the environment variable `DefaultDir` in the `[Options]` group or `WorkDir` in the `[WorkingDirectory]` group.

4.3.1 Configuring for Use with Editors

Perform the following steps to define an editor. Additional information about the **MCUez Shell** and **Configuration** dialog settings are explained in the *MCUez Installation and Configuration User's Manual*, Motorola document order number MCUEZINS/D.

1. Start the **MCUez Shell**.
2. Click on the **ezMCU** icon (first icon) in the shell. The **Configuration** dialog is displayed.
3. Click **Change ...** to open the **Current Configuration** dialog.
4. Select the **Editor** tab.
5. In the **Editor** list box, select an editor.
6. For Codewright (from Premia Corporation) or WinEdit™, enter a filename in the **Project File** edit box. Codewright project files have the extension *.pjt* and WinEdit files have the extension *.wpj*. The editor project file is created automatically in the project directory.
7. In the **Executable** edit box, enter or browse to the editor's executable file.

4.4 Automating the MCUez Startup Process

Often the same tasks have to be performed after starting the debugger. Tasks can be automated by writing a command file that contains all commands to be executed after startup. Most target components will execute the command file *startup.cmd* once the target component is loaded and initialized. By inserting the call command in the startup command file (for example, *call init.cmd*), the user-defined command file (*init.cmd*) also will be executed when the target component is loaded.

Example of *startup.cmd* file:

```
call init.cmd
...
```

Example of *init.cmd* file:

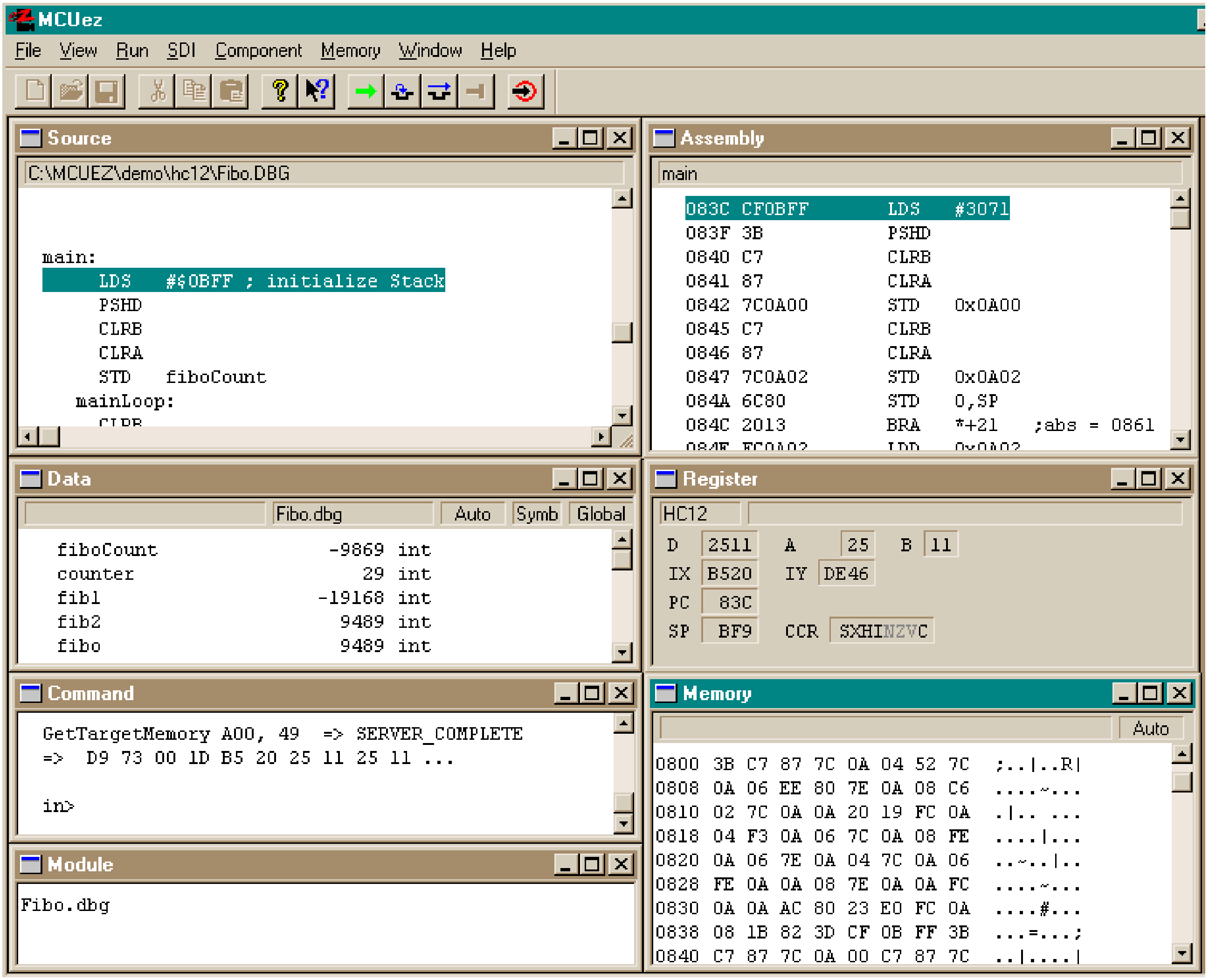
```
load fibo.abs
bs &main t
g
```

The *init.cmd* file will load the application *fibo.abs*, then set a temporary breakpoint at the start of the function `main` and start the application. The application will then stop at `main` after executing the startup and initialization code.

4.5 Loading an Application

Follow these steps to load an application:

1. Select **Load** in the target menu (for instance, SDI) to open the **Load Executable File** dialog box. The target menu is located between the **Run** and **Component** menus.
2. Select an application (for example, *fibo.abs*).
3. Click **OK** to close the dialog box and load the application into the debugger.



```
C:\MCUEZ\demo\hc12\Fibo.DBG

main:
        LDS    #$0BFF ; initialize Stack
        PSHD
        CLRB
        CLRA
        STD    fiboCount
     mainLoop:
        CLRB
```

```
main
083C CF0BFF        LDS    #3071
083F 3B            PSHD
0840 C7            CLRB
0841 87            CLRA
0842 7C0A00        STD    0x0A00
0845 C7            CLRB
0846 87            CLRA
0847 7C0A02        STD    0x0A02
084A 6C80          STD    0,SP
084C 2013          BRA    *+21    ;abs = 0861
084F FC0A02        LDD    0x0A02
```

```
Fibo.dbg
fiboCount              -9869  int
counter                   29  int
fib1                  -19168  int
fib2                    9489  int
fibo                    9489  int
```

```
HC12
D   2511    A    25    B  11
IX  B520    IY  DE46
PC   83C
SP   BF9    CCR  SXHINZVC
```

```
GetTargetMemory A00, 49  => SERVER_COMPLETE
=>  D9 73 00 1D B5 20 25 11 25 11 ...

in>
```

```
0800  3B  C7  87  7C  0A  04  52  7C   ;..|..R|
0808  0A  06  EE  80  7E  0A  08  C6   ....~...
0810  02  7C  0A  0A  20  19  FC  0A   .|.. ...
0818  04  F3  0A  06  7C  0A  08  FE   ....|...
0820  0A  06  7E  0A  04  7C  0A  06   ..~..|..
0828  FE  0A  0A  08  7E  0A  0A  FC   ....~...
0830  0A  0A  AC  80  23  E0  FC  0A   ....#...
0838  08  1B  82  3D  CF  0B  FF  3B   ...=...;
0840  C7  87  7C  0A  00  C7  87  7C   ..|....|
```

```
Fibo.dbg
```

Figure 4-1. Loading an Application

The source component displays the source from the module containing the entry point for the application. The highlighted statement is the entry point.

The assembly component contains the corresponding disassembled code. The highlighted statement is the entry point from the application. The code is disassembled directly from the target board memory.

The global data component contains the list of global variables defined in the module containing the application entry point.

The PC (program counter) in the register component is initialized with the initial value from the application entry point.

4.6 Starting an Application

The two ways to start an application are:

1. Select the **Run| Start/Continue** menu option.
2. Click the **Start/Continue** button in the MCUez toolbar ➜.

The message `RUNNING` in the status bar indicates that the application is running.

The application will continue until:

- It is manually halted.
- A breakpoint has been reached.
- An exception has been detected.

4.7 Stopping an Application

The two ways to stop an application are:

1. Choose **Run | Halt**.
2. Click on the **Halt** button in the MCUez toolbar . `HALTED` is displayed in the status bar to indicate that program execution has stopped.

The highlighted line in the source component is the source statement where the program was stopped (for example, the next statement that will be executed).

The highlighted line in the assembly component is the assembler statement where the program was stopped (for example, the next assembler instruction that will be executed).

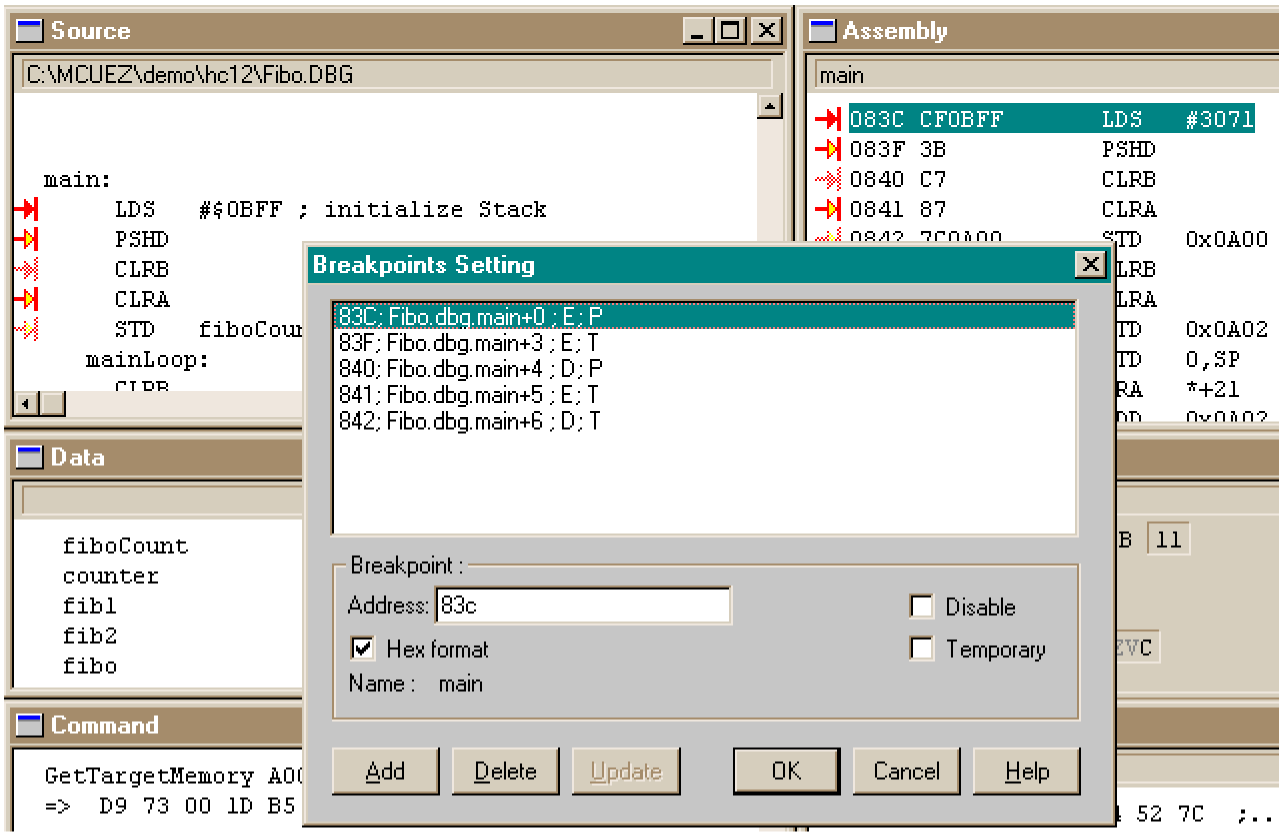
The data component displays the names and values of global variables defined in the currently executed module. The name of the module is specified in the data component information bar.

4.8 Breakpoints

Breakpoints are control points associated with a program counter (PC) value (for instance, program execution is stopped as soon as the PC reaches the value defined in a breakpoint). The MCUez debugger supports different types of breakpoints:

- Run-to-cursor breakpoints (temporary breakpoints), which are deleted as soon as they are reached. This type of breakpoint is activated the next time the corresponding instruction is executed.
- Set breakpoints (permanent breakpoints), which remain active until the user deletes them. This type of breakpoint will be activated each time the corresponding instruction is executed.

Breakpoints can be set in either a source or assembly component. **Figure 4-2** shows the **Breakpoints Setting** dialog window.



```
C:\MCUEZ\demo\hc12\Fibo.DBG

main:
      LDS    #$0BFF ; initialize Stack
      PSHD
      CLRB
      CLRA
      STD    fiboCou
  mainLoop:
```

```
main
083C  CF0BFF        LDS    #3071
083F  3B            PSHD
0840  C7            CLRB
0841  87            CLRA
0842  7C0A00        STD    0x0A00
```

```
fiboCount
counter
fib1
fib2
fibo
```

```
GetTargetMemory A0
=>  D9 73 00 1D B5
```

```
83C; Fibo.dbg.main+0 ; E; P
83F; Fibo.dbg.main+3 ; E; T
840; Fibo.dbg.main+4 ; D; P
841; Fibo.dbg.main+5 ; E; T
842; Fibo.dbg.main+6 ; D; T
```

Breakpoint :
Address: 83c
[✓] Hex format
Name : main
[ ] Disable
[ ] Temporary

Figure 4-2. Breakpoints Setting Dialog Window

The **Breakpoint Setting** dialog consists of:

- A list box which displays a list of currently defined breakpoints
- **Breakpoint:** group box which displays the address of the currently selected breakpoint, the name of the procedure in which the breakpoint has been set, the state of the breakpoint (disable or not), and type of breakpoint (temporary or permanent)
- **Delete** button: Click **Delete** to remove the currently selected breakpoint.
- **OK** button: Click **OK** to accept all modifications.
- **Cancel** button: Click **Cancel** to retain all previous settings.
- **Help** button: Click **Help** to open the help file and associated information.

The list box allows multiple consecutive breakpoints to be selected. Select a breakpoint, then hold the Shift key, and select another breakpoint.

Select multiple non-consecutive breakpoints by selecting a breakpoint, then holding the control (Ctrl) key, and selecting other breakpoints.

When selecting multiple breakpoints, the name of the group box **Breakpoint:** changes to **Selected breakpoints:** and the associated controls **Address (hex)**, and **Name:** are disabled.

4.8.1 Breakpoint Symbols

A temporary breakpoint has this symbol : .

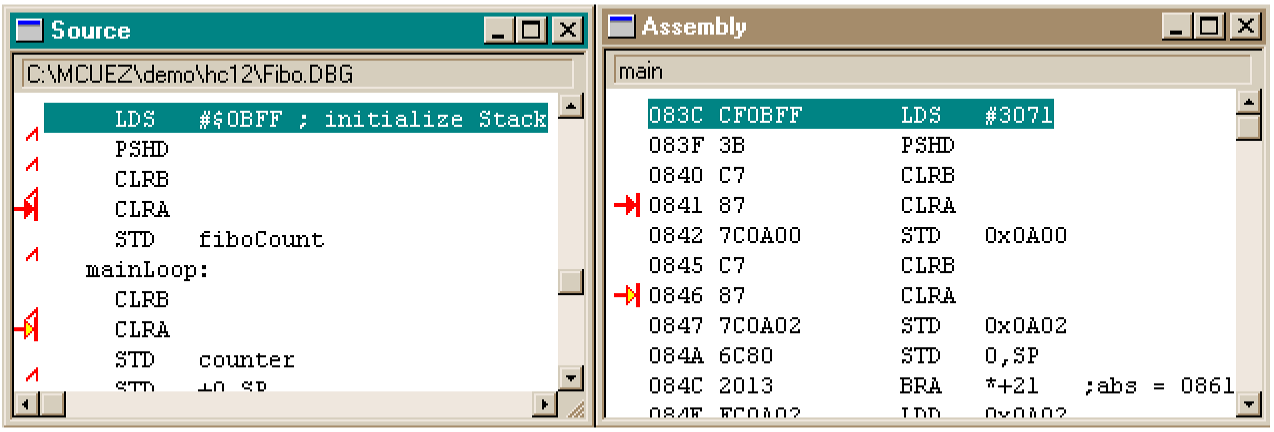
A permanent breakpoint has this symbol: .

A disabled breakpoint looks like: .

4.8.2 Identifying All Positions to Define a Breakpoint

Some compound statements (a statement that can be split into several base instructions) can be generated when using a high level language. The MCUez debugger helps detect all positions where a breakpoint can be set:

1. Right click in the source component. The source pop-up menu is displayed.
2. Check **Marks** from the pop-up menu. All statements where a breakpoint can be set are identified by a special mark: . See **Figure 4-3**.
3. To remove the breakpoint marks, right click in the source component and uncheck **Marks**.



```
C:\MCUEZ\demo\hc12\Fibo.DBG

      LDS    #$0BFF ; initialize Stack
      PSHD
      CLRB
      CLRA
      STD    fiboCount
    mainLoop:
      CLRB
      CLRA
      STD    counter
      STD    +0,SP
```

```
main

083C  CF0BFF        LDS    #3071
083F  3B            PSHD
0840  C7            CLRB
0841  87            CLRA
0842  7C0A00        STD    0x0A00
0845  C7            CLRB
0846  87            CLRA
0847  7C0A02        STD    0x0A02
084A  6C80          STD    0,SP
084C  2013          BRA    *+21    ;abs = 0861
084E  FC0A02        LDD    0x0A02
```

Figure 4-3. Identifying Breakpoint Positions

4.8.3 Defining a Breakpoint

The debugger provides two ways to define a breakpoint:

1. Using the pop-up menu:
   – Point at a statement in the **Source** or **Assembly** component window and click the right mouse button to display the pop-up menu.
   – Select **Set Breakpoint** or **Run to Cursor** from the pop-up menu. A breakpoint mark is displayed in front of the selected statement.

2. Using the keyboard:
   - Point at a statement in the **Source** component window, hold the left mouse button, and press the P key for a permanent breakpoint or T for a temporary breakpoint.
   - A breakpoint mark is displayed in front of the selected statement.

Once a breakpoint has been defined, program execution can continue. The application stops before executing the statement. Permanent breakpoints remain active until they are disabled or deleted.

4.8.4 Deleting a Breakpoint

The MCUez debugger provides four ways to delete a breakpoint.

1. Using **Delete Breakpoint** from pop-up menu:
   - In the source or assembly component, point at a statement where a breakpoint has been defined and right click.
   - Select **Delete Breakpoint** from the pop-up menu.
2. Using the keyboard:
   - In the source or assembly component, point at a statement where a breakpoint has been defined, hold down the left mouse button, and press the D key.
3. Select **Show Breakpoints...** from **Source** pop-up menu:
   - Place the mouse pointer in the **Source** or **Assembly** component window and right click.
   - Select **Show Breakpoints** from the pop-up menu. The **Breakpoints Setting** dialog is displayed.
   - In the list of defined breakpoints, select the breakpoint to delete.
   - Click **Delete**. The selected breakpoint is removed from the list of defined breakpoints.
   - Click **OK** to close the **Breakpoints Setting** dialog box.
4. Select **Run | Breakpoints ...**:
   - Choose **Run | Breakpoints ...** to display the **Breakpoints Setting** dialog.
   - Select the breakpoint to delete.
   - Click **Delete**.
   - Click **OK** to close the **Breakpoints Setting** dialog box.

4.8.5 Breakpoints Menu

Figure 4-4 shows the **Breakpoints** pop-up menu.

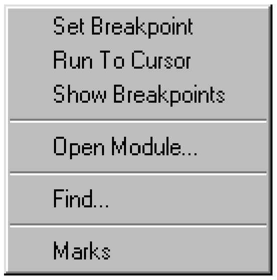


Figure 4-4. Breakpoints Menu

Table 4-1 defines all entries in the breakpoints pop-up menu.

Table 4-1. Breakpoints Pop-up Menu Definitions

Menu Entry	Description
Set Breakpoint	Selects the permanent breakpoint option
Run To Cursor	When selected, sets a temporary breakpoint at the nearest code position and continues execution of the program immediately. If a disabled breakpoint is at this position, the temporary breakpoint will also be disabled and the program will not halt. Temporary breakpoints are removed automatically when they are reached.
Show Breakpoints	Opens the **Breakpoints Setting** dialog box and allows the user to consult the list of breakpoints defined in the application and to modify their properties.
Marks	Toggles the display of source positions where breakpoints may be set. If this switch is on, source positions are marked by upside down check marks.

NOTE: *If some statements do not show marks although the mark display is switched on, the following may be at fault:*

- *The statement did not produce code due to compiler optimization.*
- *The entire procedure or section is not linked in the application because of smart linking.*

4.9 Stepping in the Application

The MCUez debugger provides stepping functions at the assembler level. The following sections describe how to implement stepping functions.

4.9.1 Stepping on Assembly Level

The debugger provides two ways of stepping to the next assembler instruction:

1. Select **Run | Assembly Step**.
2. Click the **Assembly Step** button on the toolbar: .

The message, `TRACED`, in the status line indicates that the application is stopped by an assembly step function. The application stops at the next assembler instruction.

The display in the source component is always synchronized with the display in the assembly component. The highlighted instruction in the source component is the source instruction that has generated the highlighted instruction in the assembly component.

Elements from the register, memory, or data components (displayed in red) are values that have changed during execution of the assembly instruction.

4.9.2 Stepping Over a Function Call

The debugger provides two ways of stepping over a function call:

1. Select **Run | Step Over**.
2. Click the **Step Over** button on the toolbar: .

The message, `STEPPED OVER`, in the status line indicates that the application is stopped by a step over function. If the application was stopped previously on a function invocation (a JSR (jump to subroutine) or BSR (branch to subroutine) instruction), a **Step Over** stops the application on the source instruction following the function invocation.

Elements from the register, memory, or data components (displayed in red) are values that have changed after the **Step Over** function was invoked.

4.10 Working with Variables

The following sections describe how to work with variables in the MCUez debugger.

4.10.1 Displaying Global Variables from a Module

The debugger provides two ways to view a list of global variables defined in a module:

1. Using drag and drop:
   - Drag a module name from the module component to a data component.
2. Using pop-up menu:
   - Place the mouse pointer in the data component and right click.
   - Select **Open Module** in the pop-up menu. A dialog box that contains the list of all modules used to build the application opens.
   - Double click on a module name. The data component with a global attribute that is neither frozen nor locked is the destination component.

The destination data component displays the list of variables and their values that are defined in the selected module.

4.10.2 Changing the Variable Value Display Format

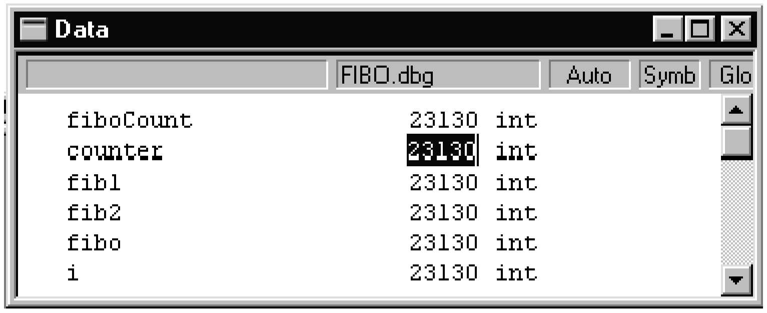
The debugger can display variable values in different formats. The **Format** entry in the pop-up menu provides several options. The selected format affects all data component variables. See **Table 4-2**.

Table 4-2. Changing the Variable Value Display Format

Menu Entry	Description
Hex	Variable values are displayed in hexadecimal format.
Oct	Values are displayed in octal format.
Dec	Values are displayed in signed decimal format.
UDec	Values are displayed in unsigned decimal format.
Bin	Values are displayed in binary format.
Symbolic	Displayed format depends on the variable type. Values for character variables are displayed in ASCII character and decimal format. Values for other variables are displayed in signed or unsigned decimal format depending on the variable being signed or not.

4.10.3 Modifying a Variable Value

The debugger allows variable values to be changed. Double click on a variable. The current value is highlighted and can be edited (**Figure 4-5**).



```
fiboCount        23130  int
counter          23130  int
fib1             23130  int
fib2             23130  int
fibo             23130  int
i                23130  int
```

Figure 4-5. Modifying a Variable Value

The following procedure describes how to modify a variable value:

1. Formats for the input value follow the rule for ANSI C constant values. Values are prefixed by `0x` for a hexadecimal value or `0` for octal values. All other values are treated as decimal values. For example, if the data component is in decimal format and the variable input value is 0x20, then the variable is initialized with 32. If a variable input value is 020, the variable is initialized with 16.

2. To accept the input value, press either the Enter key or Tab key. To restore the previous value, press the escape (Esc) key or select another variable before pressing the Enter or Tab keys.
3. If an input value has been validated by pressing the Tab key, the next variable value in the component is highlighted automatically.

4.10.4 Displaying an Allocated Variable Address

The start address and variable size are displayed in the data information bar when a variable name is clicked.

4.10.5 Loading an Address Register with a Variable Address

To load a register with the address of a variable, drag a variable name from the data component to the register component. The destination register is updated with the start address of the selected variable.

4.11 Working with Registers

The following sections describe how to work with registers.

4.11.1 Changing the Register Display Format

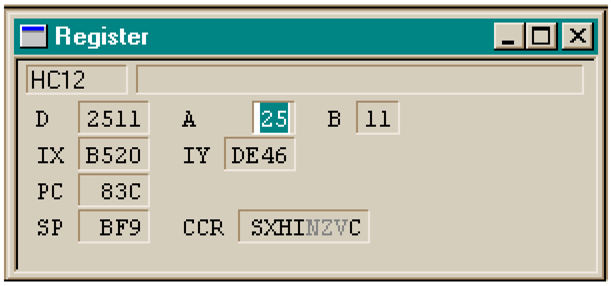
The debugger allows the register content to be displayed in hexadecimal or binary format. To do so:

1. Right click in the register component to display the pop-up menu.
2. Select **Options ...**
3. Select either binary or hexadecimal format.

4.11.2 Modifying the Content of an Index or Accumulator Register

The following procedure describes how to modify the content of an index or accumulator register. The register window changes to reflect the MCU used in specific systems.

1. Double click on a register. The current register content is highlighted (**Figure 4-6**) and can be edited.



```
Register
HC12
D   2511   A     25   B 11
IX  B520   IY  DE46
PC   83C
SP   BF9   CCR  SXHINZVC
```

Figure 4-6. Modifying the Content of an Index or Accumulator Register

2. The format of the input value depends on the format selected for the register component. If the format is hex, the input value is treated as a hex value. If the input value is 10, the variable will be set to *`0x10 = 16`*.
3. To accept the input value, press either the Enter key or Tab key or select another register. To restore the previous value, press the escape (Esc) key or select another variable before pressing the Enter or Tab keys.
4. If an input value has been validated by pressing the Tab key, the content of the next register is highlighted automatically.

4.11.3 Modifying Bit Register Contents

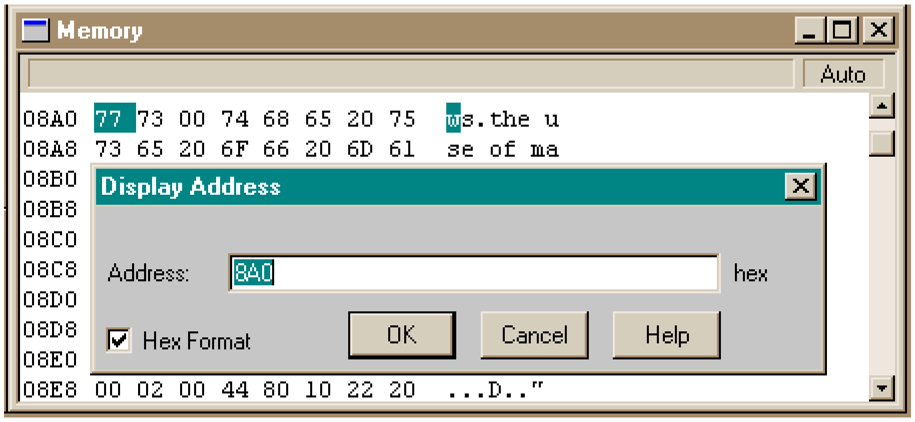
In a bit register, each bit has a specific meaning, for example, a status register (SR) or condition code register (CCR) bit from a processor. Mnemonic characters for bits set to 1 (one) are displayed in black. Mnemonic characters for bits reset to 0 (zero) are displayed in grey. Each bit inside the bit register is toggled by double clicking on the corresponding mnemonic character.

4.11.4 Retrieving a Memory Dump Starting at a Register-Indicated Address

The MCUez debugger provides two ways to dump the memory starting at the address a register is pointing to:

1. Using drag and drop:
   - Drag a register from the register component to the memory component.
2. Select **Address...** menu entry:
   - Right click in the memory component to display the pop-up menu.
   - Select **Address ...** to open the **Memory** dialog box.
   - Enter the register content in the **Address:** field and click **OK** to close the dialog box.

The memory component scrolls until it reaches the address specified in the dialog box. This feature allows the display of a memory dump from the application stack.



```
08A0 77 73 00 74 68 65 20 75  ws.the u
08A8 73 65 20 6F 66 20 6D 61  se of ma
08B0
08B8
08C0
08C8
08D0
08D8
08E0
08E8 00 02 00 44 80 10 22 20  ...D.."
```

Address: 8A0 hex
[✓] Hex Format

Figure 4-7. Choosing a Memory Address

NOTE: *If **Hex Format** is checked in the **Display Address** dialog box, numbers and letters are treated as hexadecimal numbers. Otherwise, expressions can be typed and hex numbers should be prefixed with `0x` or $.*

4.12 Working with Memory

The following sections describe how to work with and modify memory content.

4.12.1 Changing the Memory Display Format

The **Format** menu entry in the **Memory** component pop-up menu provides several options. **Table 4-3** defines the display format options.

Table 4-3. Memory Display Format Options

Menu Entry	Description
Hex	Memory dump is displayed in hexadecimal format.
Dec	Memory dump is displayed in signed decimal format.
UDec	Memory dump is displayed in unsigned decimal format.
Oct	Memory dump is displayed in octal format.
Bin	Memory dump is displayed in binary format.

4.12.2 Modifying Memory Address Content

The debugger allows the content of a memory address to be changed. To do so:

1. Double click the memory location to be modified. The content of the memory location is highlighted and can be edited.
2. The format for the input value depends on the format selected for the memory component. If the format is hex, the input value is treated as a hex value. For example, if the input value is 10, the memory address will be set to 0x10 = 16.
3. Once a value has been allocated to a memory word, it is validated and the next memory address is selected automatically.
4. To validate the new value, press Enter or Tab or select another memory position. To restore the previous value, press the escape (Esc) key before pressing the Enter or Tab key.

Section 5. Command Reference

5.1 Contents

5.2 Introduction

This section provides a detailed list of all MCUez debugger commands. All commands and component names are case insensitive. The EBNF (Extended Backus-Naur Form) command syntax is:

```
[<component name> [:<component number>] < ] <command>
```

where `component name` is the name of the component.

The `component number` is the number of the component. This number does not exist in the component window title if only one component of this type is open. When two instances of the data component are open, each data component is titled numerically as in `Data:1` and `Data:2`. A number is automatically associated with a component if more than one instance of the component exists.

The redirect left symbol (<) redirects a command to a specific component. Some commands are valid for several or all components. If the command is not redirected to a specific component, all concerned components will be affected. Also a mismatch could occur due to the fact that command parameters could differ from one component to another for the same command name.

5.3 List of Available Commands

The following sections list and define each available MCUez debugger command.

5.3.1 Kernel Commands

Kernel commands are used to build command files. Therefore, they can be used only in an MCUez command file. The command line component accepts one command at a time. It is possible to build powerful files, combining kernel commands with target commands and component commands.

Command Syntax	Short Description
`AT time`	Sets a time condition for a command execution
`ELSE`	Alternate operation associated with `IF` command
`ELSEIF condition`	Alternate conditional operation associated with `IF`
`ENDFOCUS`	Resets the current focus (see `FOCUS` command)
`ENDFOR`	Exits a `FOR` loop
`ENDIF`	Exits an `IF` condition
`ENDWHILE`	Exits a `WHILE` loop
`FOCUS componentName`	Sets the focus on a specified component
`FOR[variable =]range [“,” step]`	`FOR` loop instruction
`GOTO label`	Unconditional branch to a label in a command file
`GOTOIF condition Label`	Conditional branch to a label in a command file
`IF condition`	Conditional execution
`REPEAT`	`REPEAT` loop instruction
`RETURN`	Returns from a `CALL` command
`UNTIL condition`	Condition of a `REPEAT` loop
`WHILE condition`	`WHILE` loop instruction
`WAIT [time] [;s]`	Command file execution pause

5.3.2 Target Commands

Target commands are used to monitor the hardware target execution. Target input/output files, target execution control, direct memory editing, breakpoint management, and CPU register setup are handled by these commands. Target commands are executed independent of open components.

Command Syntax	Short Description
`BC <address | *>`	Deletes a breakpoint
`BD`	Displays list of all breakpoints
`BS <address> [P|T]`	Sets a breakpoint
`CALL [filename] [;C][;NL]`	Executes a command file
`CD [path]`	Changes the current working directory
`CF [filename] [;C] [;NL]`	Executes a command file
`CR [filename][;A]`	Opens a record file
`DASM [address|range][;OBJ]`	Disassembles
`DB [address|range]`	Displays memory bytes
`DEFINE <symbol> [=] <expression>`	Defines a user symbol
`DL [address|range]`	Displays memory bytes as longwords
`DW [address | range]`	Displays memory bytes as words
`G [address]`	Starts execution of the loaded application
`LF [filename][;A]`	Opens a log file
`LOG <type>[=]<state> {[,]<type>[=]<state>}`	Sets options for the log file
`LS [symbol | *][;(C | S)]`	Displays the list of symbols
`MEM`	Displays the memory map
`MS <range> <list>`	Sets memory bytes
`NB [base]`	Sets the base of arithmetic operations
`NOCR`	Closes the record file
`NOLF`	Closes the log file
`P [address]`	Single assembly steps into the program
`RD [list | *]`	Displays content of registers
`RS <register>[=]<value> {[,]<register>[=]<value>}`	Sets a register

Command Syntax	Short Description
`S`	Stops application execution
`SAVE <range> <filename> [<offset>][;A]`	Saves a memory block in S-record format
`SREC <filename> [<offset>]`	Loads a memory block in S-record format
`T [address][,count]`	Traces instructions at specified address
`UNDEF [symbol | * ]`	Undefines a user symbol
`WB <range> <list>`	Writes bytes
`WL <range> <list>`	Writes longwords
`WW <range> <list>`	Writes words

5.3.3 Component Commands

Component commands monitor the debugger environment, component windows, component window layouts, and load component windows and user applications.

Command Syntax	Short Description
`ACTIVATE <component name>`	Activates a component window
`ATTRIBUTES list`	Sets up the display inside a component window
`AUTOSIZE on|off`	Autosizes windows in main window
`BCKCOLOR color`	Sets the background color
`CLOSE <component name> | *`	Closes a component
`E <expression> [;(O|D|X|C|B)]`	Evaluates a given expression
`FILL <range> <value>`	Fills a memory range with a value
`FIND <string> [;B] [;MC] [;WW]`	Finds and highlights a pattern
`FONT ‘fontName’ [size][color]`	Sets text font
`HELP`	Displays a list of available commands
`LOAD applicationName`	Loads user’s application
`OPEN <component name> [x y width height][;i|;max]`	Opens a component
`SLAY <filename>`	Saves the general window layout
`SMEM range`	Shows a memory range
`SMOD module`	Shows a module
`SPC address`	Shows the specified address in a component window
`SPROC level`	Shows information associated with a specific procedure
`UPDATERATE rate`	Sets the data update mode
`VER`	Displays version number of components and MCUez
`ZOOM <address in | out>`	Zooms in/out on an array

5.4 Definition of Terms

A definition and explantion of how certain words are used in command syntax descriptions follows.

- **address** — A number matching a memory address. This number can be specified in the ANSI C format (for instance, `0x` for hexadecimal value, O for octal) or in the MCUez assembler format ($ for hexadecimal, @ for octal, % for binary).

  Example: `255, 0377, 0xFF, $FF`

NOTE: `address` *can also be an "expression" if "constant address" is not specifically mentioned in the command description. An "expression" can be: Global application variables, I/O register definitions defined in* `DEFAULT.REG`*, definitions in the command line, and numerical constants.*

```
Example:  DEFINE IO_PORT = 0x210
              WB IO_PORT 0xFF
```

- **range** — A composition of two addresses that define a memory address range. The syntax is shown as:

  `address..address`

  or

  `address, size`

  where `size` is an ANSI format numerical constant.

  Example: `0x2F00..0x2FFF`

  Refers to a memory range starting at `0x2F00` and ending at `0x2FFF` (256 bytes).

  Example: 0x2F00,256

  Refers to a memory range starting at `0x2F00`, which is 256 bytes wide. Both previous examples are equivalent.

- **filename** — A DOS filename and path that identifies a file and its location. The command interpreter does not assume the filename extension. Use backslash (\) or slash (/) as a directory delimiter. The parser is case insensitive. If no path is specified, it looks for the file or writes the file into the current project directory; for instance, when no path is specified, the default directory is the project directory.

- **component** — Name of a component window

  Example: `Memory`

5.5 Register Description File

When loading an MCUez target, the definition of the I/O registers is loaded from a file. This allows the names of these registers to be used as parameters for commands or as operands in an expression. The syntax of the file is defined in the **Appendix C. Extended Backus-Naur Form (EBNF)**.

There may be several different files depending on the MCU used. The name of the correct file is derived from the MCU identification number (MCU Id) in the following way:

Mcuioxxx.reg

xxx is the MCU Id in hexadecimal representation. This file is expected to be found in the directory where the program files are located (for instance, ..*\PROG\REG*). If this file is not found, corresponding information will be missing and related commands may not deliver the complete results.

5.5.1 File Format

A header contains the name, identification number, and location of the register block of the MCU. The header is followed by a list of module descriptors. Each descriptor contains register definitions and (optionally) a memory map specification. The register definitions can be grouped under a group name. Each register definition defines the name, address, and size of an I/O register. The memory map specification is used by the `MEM` command to display the configured memory of that module.

5.6 Expressions

Many commands accept expressions as parameters. Expression syntax and semantics descriptions follow.

5.6.1 Expression Definition in EBNF

Example:

```
expression = lorExpr
lorExpr    = landExpr {"||" landExpr} // logical OR
landExpr   = orExpr {"&&" orExpr}     // logical AND
orExpr     = xorExpr {"|" xorExpr}    // bitwise OR
xorExpr    = andExpr {"^" andExpr}    // bitwise XOR
andExpr    = eqExpr {"&" eqExpr}      // bitwise AND
eqExpr     = relExpr {("==" | "!=") relExpr}
relExpr    = shiftExpr {("<" | ">" | "<=" | ">=")
             shiftExpr}
shiftExpr  = addExpr {("<<" | ">>") addExpr}
addExpr    = mulExpr {("+" | "-") mulExpr}
MulExpr    = castExpr {("*" | "/" | "%") castExpr
castExpr   = ["~" | "!" | "+" | "-" ] parenExpr
parenExpr  = "(" expression ")"
             | cObject
             | symbol
             | register
             | variable
             | string
             | number
symbol defined with the DEFINE command found
in ANSI C
register    = IOReg
variable   = ObjectReg
ObjectReg  = ["OBJPOOL::"] ObjectSpec
ObjectSpec = ObjectName ["." FieldName].
ObjectName = ident [":" Index].
FieldName  = IdentNum ( [".." IdentNum] | ["." Size] ).
IdentNum   = ident | "#" HexNumber.
Size       = "B" | "W" | "L".
```

`ident` is an identifier as defined in ANSI C

```
IOReg       = ["IOREG::"] group | regName
```

`group` refer to the Motorola I/O register file definition in **Appendix A. Register Description File**

`regName` refer to the Motorola register name definition in **Appendix A. Register Description File**

```
itemName    = module |[[module] ":"] procedure |
                [[module] ":" [procedure] ":"] variable
variable    = ident { "." ident | number }
module      = ident ["." extension]
procedure   = ident
```

`extension` is an identifier as defined in ANSI C
`number` is a number as defined in ANSI C
`ident` is an identifier as defined in ANSI C

Module names can have an extension. If no extension is specified, the parser will look for the first module that has the same name (without extension).

5.6.2 Semantics

A scope represents either a module or procedure. A scope is recognized by the presence of the double colon which terminates the scope. If the scope identification contains at least one colon, it is assumed to represent a procedure; otherwise, it represents a module.

Empty module or procedure names represent the current module or procedure, respectively. The current procedure is the procedure that the program counter of the debugger points to. The current module is the module that contains the current procedure.

Items are identified as absolute or relative, corresponding to the presence or absence of a scope.

An item is identified as absolute by specifying its scope, for instance, the module and/or procedure where the item is located.

An item is identified as relative if a scope is omitted. In this case, the item is assumed to be located in the current procedure.

5.6.3 Scope Examples

`fibo.dbg:Fibonacci:fib1`	Matches the local variable `fib1` of the procedure `Fibonacci` in the module `fibo.dbg`
`:main`	Matches the procedure `main` in the current module
`start12.c:_Startup`	Matches the procedure `_Startup` in the module `start12.c`
`::counter`	Matches the global variable `counter` of the current module
`:Fibonacci:fib1`	Matches the local variable `fib1` of the procedure `Fibonacci` of the current module
`fibo.dbg::counter`	Matches the global variable `counter` of the module `fibo`
`fib1`	Matches the local variable of the current procedure or a global variable of any module
`startupData.flags`	Matches the field flags of the local or global variable `startupData` (which is a structure) of the current module or procedure

5.6.4 Constant Standard Notation

Inside an expression, the ANSI C standard notation for constant is supported. This means that independent of the current number base, hexadecimal or octal constants can be specified using standard ANSI C notation.

Example:

Notation	**Meaning**
`0x----`	Hexadecimal constant
`0----`	Octal constant

Similarly, the assembler notation for constant is supported. This means that independent of the current number base, hexadecimal, octal, or binary constants can be specified using the assembler prefixes.

Example:

Notation	**Meaning**
`$----`	Hexadecimal constant
`@`	Octal constant
`%`	Binary constant

When the default number base is 16, constants starting with a letter `A`, `B`, `C`, `D`, `E,` or `F` must be prefixed by `0x` or `$`. Otherwise, the command line detects a symbol and not a number.

Example:

Notation	**Meaning**
`5AFD`	Hexadecimal constant `$5AFD`
`AFD`	Symbol name
`$AFD`	Hexadecimal constant

5.7 Kernel Commands

Kernel commands are commands that build command files. Command files can be built by combining kernel, base, common, and component-specific commands.

AT

Short description:

Time delay for executing a command in a command file

Syntax:

```
AT <time>
```

Argument:

`time`	Expression interpreted in milliseconds

Description:

The `AT` command temporarily suspends a command from executing for a specified delay in milliseconds. The delay is measured from the time the command file is started. In the event that command files are chained (one calling another), the delay is measured from the time the first command file is started. This command can be executed only from a command file. The time specified is relative to the start of the command file.

Example:

```
    AT 10 OPEN Command
```

This command opens a command line component 10 ms after execution of the command file.

CALL

Short description:

Executes a command file

Syntax:

```
CALL [FileName] [;C][;NL]
```

Description:

The `CALL` command is an alias of the `CF` command. Refer to the `CF` command in the base commands section for a description and examples.

DEFINE

Short description:

Defines a user symbol

Syntax:

```
DEFINE symbol [=] expression
```

Arguments:

`symbol`	User-defined name
`expression`	User-defined expression assigned to symbol name

Description:

The `DEFINE` command creates a symbol and associates the value of an expression with the symbol. Arithmetic expressions are evaluated when the command is interpreted. The symbol represents the expression until the symbol is redefined or undefined using the `UNDEF` command. A symbol is a maximum of 31 text characters. In a command line, all symbol occurrences (after the command name) are substituted by their values before processing. A symbol cannot represent a command name. A symbol definition precedes (and therefore conceals) a program variable with the same name. Defined symbols remain valid when a new application is loaded.

Use this command to assign meaningful names to expressions that are used in other commands. This increases the readability of command files and avoids re-evaluation of complex expressions. An application variable or I/O register can be overwritten with a `DEFINE` command.

Example:

```
DEFINE addr $1000
DEFINE limit = addr + 15
```

First `addr` is defined as a constant equivalent to `$1000`. Then `limit` is defined and assigned the value (`$1000 + 15`). A symbol can be redefined on the command line using the `DEFINE` command. The original value of the symbol defined in the application is not accessible until an `UNDEF` is issued on the symbol name.

Example:

The symbol named `testCase` is defined in the application test.

```
/* Loads application test.abs */
LOAD   test.abs
/* Display value of the variable testCase from
the loaded*/
/* application. */
DB     testCase
/* Redefine symbol testCase. */
DEFINE testCase = $800
/*Display value stored at address $800.*/
DB     testCase
/* Undefine symbol testCase. */
UNDEF testCase
/* Display value of the variable testCase from
the loaded*/
/* application. */
DB     testCase
```

ELSE

Short description:

Alternate operation associated with `IF` command

Syntax:

```
ELSE
```

Description:

The `ELSE` keyword is associated with the `IF` command.

ELSEIF

Short description:

Alternate conditonal operation associated with `IF` command

Syntax:

```
ELSEIF condition
```

Argument:

`condition`	User-defined code

Description:

The `ELSEIF` keyword is associated with the `IF` command.

ENDFOCUS

Short description:

Resets the current focus (refer to `FOCUS` command)

Syntax:

```
ENDFOCUS
```

Description:

The `ENDFOCUS` command resets the current focus. It is associated with the `FOCUS` command. The following commands are broadcast to all currently open components. This command is only valid in a command file.

Example:

```
FOCUS Assembly
ATTRIBUTES code on
ENDFOCUS
FOCUS Source
ATTRIBUTES marks on
ENDFOCUS
```

The `ATTRIBUTES` command is first redirected to the assembly component by the `FOCUS Assembly` command. The code is displayed next to assembly instructions. Then the assembly component is released by the `ENDFOCUS` command and the second `ATTRIBUTES` command is redirected to the source component by the `FOCUS Source` command. Marks are displayed in the source window.

ENDFOR

Short description:

End of a `FOR` loop

Syntax:

```
ENDFOR
```

Description:

The `ENDFOR` keyword is associated with the `FOR` command and terminates a `FOR` loop.

ENDIF

Short description:

End of an `IF` condition

Syntax:

```
ENDIF
```

Description:

The `ENDIF` keyword is associated with the `IF` command and terminates a conditional block.

ENDWHILE

Short description:

End of a `WHILE` loop

Syntax:

```
ENDWHILE
```

Description:

The `ENDWHILE` keyword is associated with the `WHILE` command and terminates a `WHILE` loop.

FOCUS

Short description:

Sets the focus on a specified component

Syntax:

`FOCUS component`

Argument:

`component`	Component window

Description:

The `FOCUS` command sets the given component (`component`) as the destination for all subsequent commands up to the next `ENDFOCUS` command. The focus command eliminates having to repeatedly specify the same command redirection, especially in the case where command files are edited manually. It is not possible to visually notice that a component is "`FOCUS`'ed". Use the `ACTIVATE` command to activate a component window. This command is valid only in a command file.

Example:

```
FOCUS Assembly
ATTRIBUTES code on
ENDFOCUS
FOCUS Source
ATTRIBUTES marks on
ENDFOCUS
```

The `ATTRIBUTES` command is first redirected to the assembly component by the `FOCUS Assembly` command. The code is displayed next to assembly instructions. Then the assembly component is released by the `ENDFOCUS` command and the second `ATTRIBUTES` command is redirected to the source component by the `FOCUS Source` command. Marks are displayed in the source window.

FOR

Short description:

FOR loop instruction

Syntax:

`FOR[variable =]range [“,” step]`

Arguments:

`variable`	Name of a defined variable. During execution of the loop, the iteration value is stored in `variable`.
`range`	Address range constant that specifies the start and end condition for the loop
`step`	Constant number defining the increment for the iteration value

Description:

The `FOR` loop allows all commands to be executed, up to the trailing `ENDFOR`, a predefined number of times. The bounds of the range and optional steps are evaluated only at the beginning. A variable (either a symbol or a program variable) may be optionally specified, which is assigned to all values in the range during execution of the `FOR` loop. If a variable is used, it must be defined with a `DEFINE` command before executing the `FOR` command.

Assignment happens immediately before comparing the iteration value with the upper boundary. The variable is a copy of the internal iteration value. Modifications on the variable do not impact the number of iterations.

This command is halted by pressing the Esc key.

Example:

```
DEFINE loop = 0
FOR loop = 1..6,1
T
ENDFOR
```

The trace command (`T`) is performed six times.

GOTO

Short description:

Unconditional branch to a label in a command file

Syntax:

```
GOTO <Label>
```

Argument:

`Label`	User-defined label used to mark a place in code

Description:

The `GOTO` command diverts command file execution to the command that follows `Label`. `Label` must be defined in the current command file. The `GOTO` command fails if `Label` is not found. A label can be followed on the same line only by a comment.

No MCUez command is allowed on the same line as a label.

Example:

```
GOTO MyLabel
...
...
MyLabel:  // comments
```

When the instruction `GOTO MyLabel` is reached, the program pointer jumps to `MyLabel` and follows program execution from this position.

GOTOIF

Short description:

Conditional branch to a label in a command file

Syntax:

```
GOTOIF <condition> <Label>
```

Arguments:

`condition`	User-defined expression
`label`	User-defined label used to mark a place in code

Description:

The `GOTOIF` command diverts execution of the command file to the command line that follows the label if the condition is true; otherwise, execution continues on the next line in the command file. The `GOTOIF` command fails if the condition is true and the label is not found.

A label can be followed on the same line only by a comment. No MCUez command is allowed on the same line as a label.

Example:

```
DEFINE jump = 0
...
DEFINE jump = jump + 1
...
GOTOIF  jump == 10 MyLabel
T
...
MyLabel:  // comments
```

The program pointer jumps to `MyLabel` only if `jump == 10`. Otherwise, the next instruction `T` (trace command) is executed.

IF

Short description:

Conditional execution in a command file

Syntax:

`IF condition`

Argument:

`condition`	User-defined expression

Description:

The conditional commands (`IF`, `ELSEIF`, `ELSE`, and `ENDIF` subcommands) allow different command sections to be executed depending on the result of the corresponding conditions.

Conditional blocks may be nested. A conditional block can be specified to start inside an `IF`, `ELSEIF`, or `ELSE` command block.

The conditions of the `IF` and `ELSEIF` commands encompass all commands up to the next `ELSEIF`, `ELSE`, or `ENDIF` command on the same nesting level. The `ELSE` command encompasses all commands up to the next `ENDIF` command on the same nesting level.

Example:

```
DEFINE jump = 0
...
DEFINE jump = jump + 1
...
IF  jump == 10
  T
  DEFINE jump = 0
ELSEIF jump == 100
  DEFINE jump = 1
ELSE
  DEFINE jump = 2
ENDIF
```

The `jump == 10` condition is evaluated as in ANSI C and depending on the test result, the trace instruction (`T`) is executed or the `ELSEIF jump == 100` test is evaluated.

REPEAT

Short description:

REPEAT loop instruction

Syntax:

```
REPEAT
```

Description:

The REPEAT command enables a command sequence to be executed until a specified condition is true. REPEAT blocks can be nested. A REPEAT block can be started inside a REPEAT block.

Example:

```
DEFINE var = 0
...
REPEAT
  DEFINE var = var + 1
  ...
UNTIL var == 2
```

The REPEAT-UNTIL loop is identical to the ANSI C loop. The operation DEFINE var = var + 1 is executed twice, then var == 2 is executed and the loop exits.

This command can be halted by pressing the Esc key.

RETURN

Short description:

Returns from a `CALL` or `CF` command

Syntax:

```
RETURN
```

Description:

The `RETURN` command terminates the current command processing level. If executed within a command file, control is returned to the caller of the command file (for example, the first instance which did not chain execution).

Example:

in file d:\demo\cmd1.txt:

```
...
CALL d:\demo\cmd2.txt
T
...
```

Example:

in file d:\demo\cmd2.txt:

```
...
...
RETURN // returns to the caller
```

The command file *cmd1.txt* calls a second command file *cmd2.txt*. It is necessary to insert the `RETURN` instruction to return to the caller file. Then the trace instruction (`T`) is executed.

UNDEF

Short description:

Undefines a user-defined symbol

Syntax:

```
UNDEF <symbol | *>
```

Arguments:

`symbol`	User-defined symbol defined by the `DEFINE` command
`*`	If `*` is specified, all symbols previously defined by the `DEFINE` command are undefined.

Description:

The `UNDEF` command removes a symbol definition from the symbol table. `UNDEF` does not undefine symbols that have been defined in the loaded application.

Program variables whose names were redefined using the `DEFINE` command become visible again.

Undefining an undefined symbol is not considered an error.

Example:
```
DEFINE test = 1
...
UNDEF test
```

When the `test` variable is no longer needed, it can be undefined and removed from the list of symbols. After `UNDEF test`, the `test` variable cannot be used unless it is redefined.

Example:

The value of a user-defined symbol can be changed by applying the `DEFINE` command again. In this case, the previous value is replaced and lost. It is not put on a stack. Then when `UNDEF` is applied to the symbol, it no longer exists even if the value of the symbol has been replaced several times.

```
in>UNDEF *
in>DEFINE apple 0
in>LS
apple          0x0 (0)    // apple is equal to 0
in>DEFINE apple = apple + 1
in>LS
apple          0x1 (1)    // apple is equal to 1
```

```
in>DEFINE apple = apple + 1
in>LS
apple          0x2 (2)    // apple is equal to 2
in>UNDEF apple
in>LS      // apple does not exist
```

In the next example, assume that the *fibo.abs* sample is loaded. At the beginning, no user symbol is defined.

```
in>UNDEF *
in>LS
User Symbols:    // there is no user symbol
Application Symbols:    // symbols of the loaded
application
fiboCount    0x800 (2048)
counter      0x802 (2050)
_startupData 0x84D (2125)
Fibonacci    0x867 (2151)
main         0x896 (2198)
Init         0x810 (2064)
_Startup     0x83D (2109)
in>DEFINE counter = 1
in>LS
User Symbols: // there is one user symbol: counter
counter      0x1 (1)
Application Symbols: // symbols of the loaded
application
fiboCount    0x800 (2048)
counter      0x802 (2050)
_startupData 0x84D (2125)
Fibonacci    0x867 (2151)
main         0x896 (2198)
Init         0x810 (2064)
_Startup     0x83D (2109)
in>undef counter
in>LS
User Symbols: // there is no user symbol
Application Symbols: // symbols of the loaded
application
fiboCount    0x800 (2048)
counter      0x802 (2050)
_startupData 0x84D (2125)
Fibonacci    0x867 (2151)
main         0x896 (2198)
Init         0x810 (2064)
_Startup     0x83D (2109)
```

When the first `LS` command is issued, `counter` has the value 0x802.
After execution of the command `DEFINE counter =1`, `counter` takes the value 1.
After execution of the command `UNDEF counter`, `counter` has the value 0x802 again, the value it held before the `DEFINE` command was issued.

UNTIL

Short description:

Specifies the condition of a `REPEAT` loop in a command file

Syntax:

```
UNTIL <condition>
```

Argument:

`condition`	User-defined expression

Description:

The `UNTIL` keyword is associated with the `REPEAT` command.

Refer to the **REPEAT** command.

WAIT

Short description:

Insert a pause in a command file execution

Syntax:

```
WAIT [time] [;s]
```

Arguments:

`time`	Delay time in tenths of a second
`;s`	Pauses execution of command file until target is halted

Description:

The `WAIT` command pauses command file execution for a specified `time` in tenths of a second or until the target is halted when `;s` is specified.

When no parameter is specified, the command file pauses for five seconds.

If `time` is specified, the command file pauses for the specified time interval.

If `;s` is specified, the command file is paused until the target is halted (on a breakpoint, exception, etc.). If the target is already halted, the command file continues execution immediately.

If `time` and `;s` are both specified, `time` is used as a timer. The command file pauses until the target is halted. If the target is not halted within the specified time interval, execution continues as soon as the specified time elapses. If the target is already halted, the command file continues immediately.

Example:

```
WAIT 100
T
...
```

Pauses for 10 seconds before executing the trace (`T`) instruction.

WHILE

Short description:

`WHILE` loop instruction

Syntax:

```
WHILE condition
```

Argument:

`condition`	User-defined expression

Description:

The `WHILE` command executes a sequence of commands as long as `condition` is true.

`WHILE` blocks can be nested.

This command can be stopped by pressing the Esc key.

Example:

```
DEFINE jump = 0
...
WHILE jump < 100
  DEFINE jump = jump + 1
ENDWHILE
T
...
```

While `jump < 100`, the jump variable is incremented by the expression:

`DEFINE jump = jump + 1`.

When the loop is exited, the trace (`T`) instruction is executed.

5.8 Target Commands

Target commands monitor target execution. Target input/output files, target execution control, direct memory editing, and CPU register setup are handled by these commands.

BC

Short description:

Deletes a breakpoint. BC stands for breakpoint clear.

Syntax:

```
BC <address|*>
```

Arguments:

`address`	Address of breakpoint to be deleted
`*`	Deletes all breakpoints

Description:

`BC` deletes a breakpoint at the specified address. This address must be in ANSI or MCUez assembler format. The address can be replaced by an expression as shown in the example.

When `*` is specified, `BC` deletes all breakpoints.

Example:

```
BC 0x8000
```

This command deletes the breakpoint set at the address `0x8000`. The breakpoint symbol is removed from the **Source** and **Assembly** windows and from the breakpoint list.

NOTE: *Correct module names (for example,* `fibo.dbg`*) are displayed in the module component window.*

```
BC &FIBO.DBG:Fibonacci
```

In this example, an expression replaces the address. `fibo.dbg` is the module name and `Fibonacci` is the function where the breakpoint is cleared. This example deletes the breakpoint set at the start address of the symbol `Fibonacci`, defined in the module `fibo.dbg`.

Equivalent operation:

Point to the breakpoint in the **Assembly** or **Source** component window, click right mouse button, and choose **Delete Breakpoint** in the pop-up menu.

BD

Short description:

Displays a list of all breakpoints currently defined. BD stands for breakpoint display.

Syntax:

`BD`

Description:

In the command line component, the `BD` command displays a list of all breakpoints with addresses and types (temporary, permanent).

For each breakpoint, the following information is displayed:

```
<SymbolName> <address> <type>
```

`SymbolName` is the name of the symbol (or function) where the breakpoint is defined.

`address` is the address where the breakpoint is set.

`type` is the type of breakpoint. `T` stands for temporary breakpoints and `P` for permanent breakpoints.

Example:

```
in>BD
Fibonacci 0x805c T
Fibonacci 0x8072 P
Fibonacci 0x8074 T
main 0x8099 T
```

Currently, one permanent and two temporary breakpoints are set in the function `Fibonacci`, and one temporary breakpoint is set in the `main` function.

NOTE: *This list will not display whether a breakpoint is disabled or active.*

BS

Short description:

Sets a breakpoint. BS stands for breakpoint set.

Syntax:

```
BS address [P|T]
```

Arguments:

`address`	Address in which to set a breakpoint
`P`	Specifies a permanent breakpoint
`T`	Specifies a temporary breakpoint

Description:

BS sets a temporary (T) or permanent (P) breakpoint at the specified address. If no P or T is specified, the default is a permanent (P) breakpoint. The address can be specified in ANSI C or MCUez assembler format. The address can also be replaced by an expression as shown in the example.

Example:

```
BS 0x8000 T
```

This command sets a temporary breakpoint at the address `0x8000`.

```
BS $8000 P
```

This command sets a permanent breakpoint at the address `0x8000`.

```
BS &FIBO.DBG:Fibonacci
```

In this example, an expression replaces the address. `fibo.dbg` is the module name and `Fibonacci` is the function where the breakpoint is set.

NOTE: *Correct module names (for example, `fibo.dbg`) are displayed in the module component window.*

The example above sets a BP on the symbol `Fibonacci` defined in `fibo.dbg`.

Equivalent operation:

Point to a statement directly in the assembly or source component window, right click, and choose **Set Breakpoint** in the pop-up menu.

CD

Short description:

Changes the current working directory

Syntax:

```
CD [path]
```

Argument:

`path`	Path to a new working directory

Description:

The `CD` command changes the current working directory to the directory specified in `path`. When the command is entered with no parameter, the current directory is displayed.

The directory specified in the `CD` command must be a valid directory. It should exist and be accessible from the `PC`. When specifying a relative path in the `CD` command, make sure the path is relative to the current project directory.

NOTE: *When no path is specified, the default directory is the project directory.*

Example:

```
in>cd
C:\mcuez\demo
in>cd ..\prog
C:\mcuez\prog
```

The new project directory is `C:\mcuez\prog`.

CF

Short description:

Executes another command file

Syntax:

```
CF [filename] [;C] [;NL]
```

Arguments:

`filename`	Name of command file to be called to execute its commands
`;C`	Terminates command file after called command file has executed its commands
`;NL`	Commands in the called file are not logged in the **Command Line** window

Descriptions:

The `CF` command enables commands in the specified command file to be executed. The command file contains ASCII text commands.

Command files can be nested. `CF` (or `CALL`) can be used in a command file to start another command file. By default, once execution of the called command file is complete, the remaining commands in the calling file are executed. If the option `;C` is specified, the calling file terminates as soon as the called command file finishes execution. Commands following the `CALL` or `CF` command in the calling file are not executed.

When the option is omitted, remaining commands in the calling file are executed after commands in the called file have been executed.

Any error halts execution of `CF` file commands.

If the command is entered with no parameter, the **Open File** dialog is displayed. Use this dialog to select the command file to execute.

Example:

```
    in > CF commands.txt
```

The *commands.txt* file is executed from the working directory. The command file must contain MCUez debugger commands.

without `;C` option:

If a *command1.txt* file contains:

```
bckcolor green
cf command2.txt
bckcolor white
```

If a *command2.txt* file contains:

```
bckcolor red
```

Enter command:

`in>`*cf command1.txt*

// executing *command1.txt*

!bckcolor green

!cf *command2.txt*

// executing *command2.txt*

1!bckcolor red

done *command2.txt*

// resume executing *command1.txt*

!bckcolor white

done *command1.txt*

with `;C` option:

If *command1.txt* file contains:

```
bckcolor green
cf command2.txt ;C
bckcolor white
```

If *command2.txt* file contains:

```
bckcolor red
```

Enter command:

`in>`*cf command1.txt*
// executing *command1.txt*
!bckcolor green
!cf *command2.txt* `;C`
// executing *command2.txt*
1!bckcolor red
1!
1!
done *command2.txt*

done *command1.txt*

CR

Short description:

Opens a record file

Syntax:

```
CR [filename][;A]
```

Arguments:

`filename`	Name of record file. If file is not specified, a standard **Open File** dialog is displayed.
`;A`	Opens file in append mode. Commands are recorded and appended to the end of an existing record file. If the `;A` option is omitted and `filename` is an existing file, the file is cleared before records are written to it.

Description:

The `CR` command archives executed commands. Commands are listed in the specified or selected file. Commands are recorded until a close record file (`NOCR`) command is executed.

Example:

```
   in>cr /mcuez/demo/myrecord.txt ;A
```

The *myrecord.txt* file is opened in append mode.

If no path is specified, the path is assumed to be the current working directory.

DASM

Short description:

Disassembles source code

Syntax:

```
DASM [<address>|<range>][;OBJ]
```

Arguments:

`address`	Constant expression representing the address where disassembly begins
`range`	Address range constant that specifies the addresses to be disassembled. When range is omitted, a maximum of 16 instructions are disassembled.
`;OBJ`	Displays assembler code in hexadecimal

Description:

The `DASM` command displays the disassembled code of an application, starting at the address given as a parameter. When `address` and `range` are both omitted, disassembly begins at the address of the instruction that follows the last instruction disassembled by the previous `DASM` command. If this is the first `DASM` command of a session, disassembly occurs at the current address in the program counter.

Press the Esc key to stop this command.

Command line example:

```
in>DASM 0x8000
LDX     0x8045
LDY     0x8043
BEQ     *+18     ;abs = 8018
PSHY
LDY     2,X+
LDD     2,X+
CLR     1,Y+
SUBD    #1
BNE     *-5      ;abs = 800D
PULY
```

The disassembled instructions are displayed in the **Command Line** component window.

Equivalent operation:

Right click in the **Assembly** component window, select **Address...** and enter the address to start disassembly in the **Show PC** dialog.

DB

Short description:

Displays memory bytes

Syntax:

```
DB [<address>|<range>]
```

Arguments:

`address`	Constant expression representing the address to be displayed
`range`	Memory address range to display

Description:

The `DB` command displays hexadecimal and ASCII byte values for a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first byte displayed in the line, followed by the number of specified hexadecimal byte values. The hexadecimal byte values are followed by the corresponding ASCII characters, separated by spaces. Between the eighth and ninth values, a hyphen (-) replaces the space character as the separator. Each non-displayable character is represented by a period (.).

When `address` and `range` are both omitted, the first byte displayed is taken from the address following the last memory position displayed by the most recent `DB`, `DW`, or `DL` command or from address `0x0000` (for the first `DB`, `DW`, `DL` command entered).

This command can be halted by typing the Esc key.

Example:

```
in>DB 0x8000..0x800F
8000: FE 80 45 FD 80 43 27 10-35 ED 31 EC 31 69 70 83
```

```
in>DB 0x8000,8
8000: FE 80 45 FD 80 43 27 10
```

Memory bytes are displayed with matching ASCII characters.

The following example displays the byte at the address of the TCR I/O register. I/O registers are defined in a *mcuioxxx.reg* file.

Example:

```
in>DB &TCR
0012: 5A Z
```

DL

Short description:

Displays memory bytes as longword

Syntax:

```
DL [<address>|<range>]
```

Arguments:

`address`	Constant expression representing the address to be displayed
`range`	Memory address range to display

Description:

The `DL` command displays the hexadecimal values of the longwords in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first longword displayed in the line, followed by the number of specified hexadecimal longword values.

When a size is specified in the range, this size represents the number of longwords to be displayed in the command line window.

When `address` and `range` are both omitted, the first longword displayed is taken from the address following the last memory position displayed by the previous `DB`, `DW`, or `DL` command or from address `0x0000` (for the first `DB`, `DW`, `DL` command entered).

This command can be halted by typing the Esc key.

Example:

```
in>DL 0x8000,2
8000: FE8045FD 80432710
```

The content of two longwords starting at `0x8000` is displayed as longword (4-byte) values. Memory longwords are displayed in the command line component.

DW

Short description:

Displays a word

Syntax:

```
DW [<address> | <range>]
```

Arguments:

`address`	Constant expression representing the address of the first word to be displayed
`range`	Memory address range to display

Description:

The `DW` command displays the hexadecimal values of the words in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first word displayed in the line, followed by the number of specified hexadecimal word values.

When a size is specified in the range, this size represents the number of words that should be displayed in the command line window.

When `address` and `range` are both omitted, the first word displayed is taken from the address following the last memory position displayed by the previous `DB`, `DW`, or `DL` command or from address `0x0000` (for the first `DB`, `DW`, `DL` command entered).

This command can be stopped by typing the Esc key.

Example:

```
in>dw 0x8000..0x8007
8000: FE80 45FD 8043 2710
```

The content of memory range starting at `0x8000` and ending at `0x8007` is displayed as word (2-byte) values.

```
in>DW 0x8000,4
8000: FE80 45FD 8043 2710
```

The content of four words starting at `0x8000` is displayed as word (2-byte) values. Memory words are displayed in the **Command Line** component window.

E

Short description:

Evaluates a given expression

Syntax:

`E <expression>[;O|D|X|C|B]`

Arguments:

`expression`	User-defined expression
`;O`	Octal — base 8
`;D`	Decimal — base 10
`;X`	Hexadecimal — base 16
`;B`	Binary — base 2
`;C`	Displays the value of expression as an ASCII character. That is, the remainder resulting from dividing the number by 256 is displayed.

All values are displayed in the current font. Control characters (<32) are displayed as decimal.

Description:

The `E` command evaluates an expression and displays the result in the **Command Line** component window. When the expression is the only parameter entered (no option specified), the value of the expression is displayed in the default number base. The result is displayed as a signed number in decimal format and as an unsigned number in all other formats.

Example:

```
in> define a=0x12
in> define b=0x10
in> e a+b = 34
```

The addition operation of the two previously defined variables `a` and `b` is evaluated and the result is displayed in the **Command Line** window. The output can be redirected to a file by using the `LF` command (refer to `LF` and `LOG` command descriptions).

G

Short description:

Begins execution of the currently loaded application

Syntax:

```
G [address]
```

Argument:

`address`	Address constant expression. This value is loaded into the program counter before execution starts. When no `address` is entered, the address in the program counter is not altered and execution begins at the address in the program counter.

Description:

The `G` command starts code execution in the emulated system at the current address in the program counter or at the specified address. The user can specify the program entry point, skipping execution of the previous code.

Example:

```
G 0x8000
```

Program execution is started at the address `0x8000`. *RUNNING* is displayed in the status bar. The application runs until a breakpoint is reached or manually stopped.

LF

Short description:

Opens a log file

Syntax:

```
LF [filename][;A]
```

Arguments:

`filename`	DOS filename that identifies the file or device to which the log is written. The command interpreter does not assume a filename extension.
`;A`	Opens the specified file in append mode. Logged lines are appended to the end of an existing log file.

If the `;A` option is omitted and `filename` is an existing file, the file is cleared before logging begins.

Description:

The `LF` command logs commands and responses to an external file or device. While logging remains in effect, any line that is appended to the command line component window is also written to the log file. Logging continues until a close log file (`NOLF`) command is executed. When the `LF` command is entered with no filename, the **Open File** dialog box is displayed.

Use the logging option command (`LOG`) to specify the information to be logged.

Example:

```
LF LOGFILE.TXT ;A
```

The file *logfile.txt* is opened as a log file in append mode. This example assumes the log file is in the working directory.

LOG

Short description:

Enables or disables logging of specific information

Syntax:

```
LOG <type> [=] <state> {[,] <type> [=] <state>}
```

Arguments:

`type` is one of the following types:

`CMDLINE`	Commands entered on the command line
`CMDFILE`	Commands read from a command file
`RESPONSES`	Command responses are written in the log file. Responses are results of commands. For example, for the `DB` command, the displayed memory dump is the response of the command. Protocol messages are not responses, but are controlled by the `RESPONSES` parameter.
`ERRORS`	Error messages. Errors are displayed in red in the command line component. Protocol messages are not errors.
`NOTICES`	Asynchronous event notices, such as breakpoints. Notices are displayed in green in the command line component. They denote status information returned asynchronously by the target interface.
`state`	Value is `on` or `off`.

Description:

The `LOG` command enables or disables logging of user-specified information in the command line component (and to the log file, when opened with an `LF` command).

By default, when the `LOG` command is not specified, all types are on. All information types are logged in the command line component and *log* file.

Example:

```
LOG ERRORS = OFF, CMDLINE = on
```

Error messages are not recorded in the log file. Commands entered in the command line component are recorded.

Logging `IF`, `FOR`, `WHILE`, and `REPEAT`

When command logging is enabled from a command file (command file executed with the `CF` or `CALL` command without the `NL` option), all commands executed in an `IF` block are logged. All commands in a block that are not executed because the corresponding condition is not verified are also logged but preceded with a `-` (hyphen).

Example:

When executing this command file:

```
define truth = 1
IF truth
  bckcolor blue
  at 2000 bckcolor white
else
  bckcolor yellow
  at 1000 bckcolor white
ENDIF
```

This log file is generated:

```
!define truth = 1
!IF truth
!  bckcolor blue
!  at 2000 bckcolor white
!else
!-  bckcolor yellow
!-  at 1000 bckcolor white
!ENDIF
```

When command logging is enabled from a command file, all commands executed in the `FOR` loop are logged each time they are executed.

Example:

When executing this file:

```
define i = 1
FOR i = 1..3
   ls
ENDFOR
```

This log file is generated:

```
!define i = 1
!FOR i = 1..3
!   ls
i             0x1 (1)
!ENDFOR
!   ls
i             0x2 (2)
!ENDFOR
!   ls
i             0x3 (3)
!ENDFOR
```

Also, all commands executed in a `WHILE` loop are logged each time they are executed.

Example:

When executing this file:

```
define i = 1
WHILE i < 3
   define i = i + 1
ls
ENDWHILE
```

This log file is generated:

```
!define i = 1
!WHILE i < 3
!   define i = i + 1
!  ls
i             0x2 (2)
!ENDWHILE
!   define i = i + 1
!  ls
i             0x3 (3)
!ENDWHILE
```

All commands executed in a `REPEAT` loop are logged each time they are executed.

Example:

When executing this file:

```
define i = 1
REPEAT
   define i = i + 1
ls
UNTIL i == 4
```

This log file is generated:

```
!define i = 1
!REPEAT
!   define i = i + 1
! ls
i            0x2 (2)
!UNTIL i == 4
!   define i = i + 1
! ls
i            0x3 (3)
!UNTIL i == 4
!   define i = i + 1
! ls
i            0x4 (4)
!UNTIL i == 4
```

LS

Short description:

Displays the list of symbols

Syntax:

```
LS [<symbol> | *][;(C |S)]
```

Arguments:

`symbol`	Restricted regular expression that specifies the symbol whose values are to be listed. This argument is case sensitive.
`*`	Lists all symbols
`;C`	Lists symbols in accepted format, which consists of a `DEFINE` command for each symbol
`;S`	Lists symbol table statistics following the list of symbols

Description:

In the command line component, the `LS` command lists the values of symbols defined in the symbol table and defined by the user. There is no limit to the number of symbols that can be listed. The memory size determines the symbol table size. Use the `DEFINE` command to define symbols and the `UNDEF` command to delete symbols.

Symbols that are listed with the `LS` command are split in two parts: application symbols (symbols defined in the application currently loaded) and user symbols (symbols defined on the command line using the `DEFINE` command). For application symbols, `LS` displays the address of the symbol. For user-defined symbols, `LS` displays the value of the symbol.

Example:

```
in>ls
User Symbols:
j            0x2 (2)
Application Symbols:
counter      0x80 (128)
fiboCount    0x81 (129)
j            0x83 (131)
n            0x84 (132)
fib1         0x85 (133)
fib2         0x87 (135)
fibo         0x89 (137)
Fibonacci    0xF000 (61440)
Entry        0xF041 (61505)
```

When `LS` is performed on a single symbol (for example, `in>ls counter`) that is an application variable as well as a user symbol, the application variable is displayed.

Example with `j` being an application symbol as well as a user symbol:

```
in>ls j
j         0x83 (131)
```

MEM

Short description:

Displays the memory map

Syntax:

`MEM`

Description:

The `MEM` command displays a representation of the current memory mapping of the system and the lower and upper boundaries of the internal module that contains the MCU registers.

MS

Short description:

Sets memory bytes

Syntax:

```
MS <range> <list>
```

Arguments:

`range`	Address range constant that defines the block of memory to be set to the values of the bytes in the list
`list`	List of byte values to be stored in the block of memory

Description:

The `MS` command initializes a specified block of memory to a specified list of byte values. When the `range` is wider than the `list` of byte values, the `list` of byte values is repeated as many times as necessary to fill the memory block.

When the `range` is not an integer multiple of the length of the `list`, the last copy of the `list` is truncated appropriately. This command is identical to the write bytes (`WB`) command.

Example:

```
    MS 0x1000..0x100F 0xFF
```

The memory range between addresses `0x1000` and `0x100F` is filled with the `0xFF` value.

NB

Short description:

Sets the base of arithmetic operations

Syntax:

```
NB <base>
```

Argument:

`base`	New number base: 2, 8, 10, or 16

Description:

The `NB` command changes or displays the default number `base` for the constant values in expressions. The initial default number base is 10 (decimal) and can be changed to 16 (hexadecimal), 8 (octal), 2 (binary) or reset to 10 with this command. `base` is always specified as decimal constant.

If `base` is omitted, the current default number base is displayed in the command line window.

Independent of the default base number, the ANSI C standard notation for constant is supported inside an expression. That means that independent of the current number base, hexadecimal or octal constants can be specified using standard ANSI C notation.

Notation	**Meaning**
`0x----`	Hexadecimal constant
`0----`	Octal constant

Example:

```
0x2F00, /* Hexadecimal Constant */
043,    /* Octal Constant */
255     /* Decimal Constant */
```

In the same way, assembler notation for constant is also supported. That means that independent of the current number base, hexadecimal, octal, or binary constants can be specified using the assembler prefixes.

Notation	**Meaning**
`$----`	Hexadecimal constant
`@`	Octal constant
`%`	Binary constant

Example:

```
$2F00,  /* Hexadecimal Constant */
@43,     /* Octal Constant */
255     /* Decimal Constant */
%10011  /* Binary Constant */
```

When the default number base is 16, constants starting with a letter A, B, C, D, E, or F must be prefixed either by 0x or by $. Otherwise, the command line interpreter cannot detect if an integer constant or a symbol is specified.

Example:

```
in>NB 16
```

The number base is hexadecimal.

NOCR

Short description:

Closes the record file

Syntax:

```
NOCR
```

Description:

The `NOCR` command closes the current record file. The record file is opened with the `CR` command.

Example:

```
    NOCR
```

The current record file is closed.

NOLF

Short description:

Closes the log file

Syntax:

```
NOLF
```

Description:

The `NOLF` command closes the current log file. The log file is opened with the `LF` command.

Example:

```
    NOLF
```

The current log file is closed.

P

Short description:

Steps into the program using assembly step over

Syntax:

```
P <address>
```

Argument:

`address`	Address constant expression where execution begins

If `address` is omitted, execution begins with the instruction pointed to by the current value of the program counter.

Description:

The `P` command executes a CPU instruction either at a specified address or at the current instruction (the one pointed to by the program counter). This command traces through subroutine calls, software interrupts, and operations involving the following instructions:

- Branch to SubRoutine (`BSR`)
- Long Branch to SubRoutine (`LBSR`)
- Jump to SubRoutine (`JSR`)
- SoftWare Interrupt (`SWI`)
- Repeat Multiply and Accumulate (`RMAC`)

For example, if the current instruction is a `BSR` instruction, the subroutine is executed, and execution stops at the first instruction after the `BSR` instruction. For instructions that are not in this list, the `P` and `T` commands are equivalent.

When the instruction specified in the `P` command has been executed, the software displays the content of the CPU registers, the instruction bytes at the new value of the program counter, and a mnemonic disassembly of that instruction.

Example:

```
in>P 0x2808
         pA=$B5 B=$20 CCR=$48 D=$B520 IX=$6FF1 IY=$0
         SP=$BEF
         PC=$886 PPAGE=$0 DPAGE=$0 EPAGE=$0 IP=$886
         000886 EE80 LDX 0,SP
```

Register contents are displayed and the current instruction is disassembled.

RD

Short description:

Displays register contents

Syntax:

```
RD [<list>| *]
```

Arguments:

`list`	List of registers to be displayed. Registers to be displayed are separated by a space. When `RD CPU` is specified, all CPU registers are displayed. If no CPU is loaded, No CPU loaded is displayed as an error message.
*	Lists the content of the register file that is currently loaded. The address and size of each register is displayed. If no register file is loaded, an error message is displayed: No register file loaded.

Description:

The `RD` command displays the content of specified registers. The display of a register includes both the mnemonic and the hexadecimal value of the register. If the specified register is not a CPU register, it is considered to be an I/O register. The debugger looks for the specified register in the loaded register file. This file is called *mcuioxxx.reg* (where *xxx* is a number related to the MCU).

If `list` is omitted, the list and any other parameters of the previous `RD` command are used.

For the first `RD` command of a session, all CPU registers are displayed.

Example:

```
in>RD A X
A = 0x1
X = 0xF
```

Contents of registers A and X are displayed.

Example:

`in>RD CPU` // will display all CPU registers.

RS

Short description:

Sets a register

Syntax:

```
RS <register>[=]<value>{ [,]<register>[=]<value>}
```

Arguments:

`register`	Specifies the name of a register to be changed. The register string is any of the CPU register names or the name of a register in the register file.
`value`	Integer constant expression (in ANSI C or MCUez assembler format)

Description:

The RS command places specified values into specified registers. RS is followed by register name and new value.

An equal sign (=) may be used to separate the register name from the value to be assigned to the register; otherwise, they must be separated by a space. The contents of any number of registers may be set using a single RS command. If the specified register is not a CPU register, it is considered to be an I/O register. The debugger looks for the specified register in the loaded register file. This file is called *mcuioxxx.reg* (where *xxx* is a number related to the MCU).

Example:

```
    in>rs A=$0 B=$5
```

The new content of register A is `$0` and B register is `$5`. The display in the **Register** window is updated with the new values.

S

Short description:

Stops execution of the loaded application

Syntax:

`S`

Description:

The `S` command stops execution of the application. Use the `Go` (`G`) command to start or continue execution.

Example:

```
in>s
STOPPING
HALTED
```

The current application is halted.

SAVE

Short description:

Saves a memory block in S-record format

Syntax:

```
SAVE <range> <filename> [offset][;A]
```

Arguments:

`range`	Address range constant that defines the block of memory to be saved in a Motorola S-record file
`filename`	DOS filename that specifies the file to which the records are written
`offset`	Optional offset to add or subtract from addresses when writing S-records. The default is `0x0000`.
`;A`	Appends the saved S-records to the end of an existing file. If this option is omitted and the specified file exists, the file is cleared before saving the S-records.

Description:

The `SAVE` command saves a specified block of memory to a specified file in Motorola S-record format. The memory block can be reloaded later using the load S-record (`SREC`) command.

Example:

```
SAVE 0x1000..0x2000 DUMP.SX ;A
```

The memory range `0x1000` to `0x2000` is appended to the *dump.sx* file.

NOTE: *If no path is specified, the path is assumed to be the current working directory.*

SET

Short description:

Sets a new target

Syntax:

```
SET <targetName>
```

Argument:

`targetName`	Name of target (without extension) to be set

Description:

Sets a new target for the debugger and loads the target interface component. The target file (*.tgt*) must be available in the *PROG* directory.

Example:

```
    SET D-Bug12
```

The D-Bug12 target is set in the debugger.

SREC

Short description:

Loads the S-record file in memory

Syntax:

```
SREC <filename> [offset]
```

Arguments:

`filename`	S-record file
`offset`	A signed value added to the addresses stored in the file when loading the file contents

Description:

The `SREC` command loads Motorola S-records from a specified file.

Example:

```
    SREC DUMP.SX
```

The *dump.sx* file is loaded into memory.

NOTE: *If no path is specified, the path is assumed to be the current working directory.*

T

Short description:

Traces program instructions. Program trace begins at a specified address.

Syntax:

```
T [<address>][,<count>]
```

Arguments:

`address`	Address constant expression at which execution begins. If `address` is omitted, the instruction pointed to by the current value of the program counter is the first instruction traced.
`count`	Integer constant expression, in the decimal integer interval [1, 65,535], that specifies the number of instructions to be traced. If `count` is omitted, one instruction is traced.

Description:

The `T` command executes one or more instructions starting at a specified address or at the current instruction (the address in the program counter). The `T` command traces into subroutine calls and software interrupts. For example, if the current instruction is a branch to subroutine (`BSR`), the `BSR` is traced, and execution stops at the first instruction of the subroutine. After executing the last (or only) instruction, the `T` command displays the contents of the CPU registers, the instruction bytes at the new address in the program counter, and a mnemonic disassembly of the current instruction.

This command can be stopped by pressing the Esc key.

Example:

```
in>T 0xF030
TRACED
A=0x0 HX=0x7F02 SR=0x62 PC=0xF032 SP=0x44D
00F032 B787   STA 0x87
```

Contents of registers are displayed and the current instruction is disassembled.

WB

Short description:

Sets a specified block of memory to a specified list of byte values

Syntax:

```
WB <range> <list>
```

Arguments:

`range`	Address range constant that defines the block of memory to be initialized to the values of the bytes in the list
`list`	List of byte values to be stored in the block of memory

Description:

The `WB` command initializes a specified block of memory with a specified list of byte values. When the range is wider than the list of byte values, the list of byte values is repeated as many times as necessary to fill the memory block. When the range is not an integer multiple of the length of the list, the last copy of the list is truncated accordingly. This command is identical to the memory set (`MS`) command.

Example:

```
WB 0x0401 0x0419 0x69
```

This command fills the memory range `0x0401..0x0419` with the byte value `0x69`.

Example:

```
WB 0x0205..0x0220 0xFF 0xEE 0xDD 0xCC 0xBB 0xAA
```

This command fills the memory range `0x0205..0x0220` with the byte value of the list.

WL

Short description:

Sets a specified block of memory to a specified list of longword values

Syntax:

```
WL <range> <list>
```

Arguments:

`range`	Address range constant that defines the block of memory to be initialized to the longword values in the list
`list`	List of longword values to be stored in the block of memory

Description:

The `WL` command initializes a specified block of memory with a specified list of longword values. When the range is wider than the list of longword values, the list of longword values is repeated as many times as necessary to fill the memory block. When the range is not an integer multiple of the length of the list, the last copy of the list is truncated accordingly.

Example:

```
    WL 0x2000 0x0FFFFF0F
```

This command fills the memory address `0x2000..0x2003` with the longword value `0x0FFFFF0F`.

WW

Short description:

Sets a specified block of memory to a specified list of word values

Syntax:

```
WW <range> <list>
```

Arguments:

`range`	Address range constant that defines the block of memory to be initialized to the word values in the list
`list`	List of word values to be stored in the block of memory

Description:

The `WW` command initializes a specified block of memory with a specified list of word values. When the range is wider than the list of word values, the list is repeated as many times as necessary to fill the memory block. When the range is not an integer multiple of the length of the list, the last copy of the list is truncated accordingly.

Example:

```
WW 0x2000..0x200F 0xAF00
```

This command fills the memory range `0x2000..0x200F` with the word value `0xAF00`.

5.9 Component Commands

The commands listed in this section monitor the MCUez debugger environment, component operation, component window layouts, and loads component windows.

ACTIVATE

Short description:

Activates a component window

Syntax:

```
ACTIVATE <component>
```

Argument:

`component`	Component window

Description:

`ACTIVATE` enables a component window. The window is displayed in the foreground and its title bar is highlighted.

If the component was previously iconized, it is opened and displayed in the foreground and its title bar is highlighted.

Example:

```
ACTIVATE Memory
```

This command will make the memory component the top most window and activate it.

ATTRIBUTES

Short description:

Sets the display and formatting attributes for a component window. Usually, this command is not specified interactively by the user. However this command can be written in a session record file or in a configuration file to save and reload component window layouts. An interactive equivalent operation is possible by using MCUez menus and operations (drag and drop, etc.), as described in the following equivalent operations sections.

In the Assembly Component

Syntax:

```
ATTRIBUTES <list>
```

Arguments:

`list=command{,command}`

command = `ADR ON|OFF | SMEM` range `| SPC` address `|CODE(ON|OFF)|ABSADR (ON|OFF) | TOPPC` address

`address`	Address to be located
`range`	Memory range to be located
`module`	Specified module
`CODE on`	Switches on the machine code display
`CODE off`	Switches off the display
`ADR on`	Switches on display of addresses in front of disassembly instruction
`ADR off`	Switches off display of addresses in front of disassembly instruction
`ABSADR on`	Switches on display of absolute address for destination of branch instructions
`ABSADR off`	Switches off display of absolute address for destination of branch instructions
`SPC address`	PC address location
`TOPPC address`	Address location of the first line of the PC

Description:

The `ATTRIBUTES` command sets the display and state options for the **Assembly** component window.

The `ADR` command displays or hides the address of a disassembled instruction.

`SMEM` (show memory range) and `SPC` (show PC address) scroll the assembly component to the corresponding address or range code location and select/highlight the corresponding assembler instructions or set of instructions.

The `CODE` command displays or hides the machine code of the disassembled instruction.

The `ABSADR` command shows or hides the destination absolute address in a disassembled instruction, such as branch to.

The `TOPPC` command specifies the PC of the first visible line.

Example:

```
Assembly < ATTRIBUTES ADR ON,CODE ON, SMEM
0x800,16
```

Addresses and hexadecimal codes are displayed in the **Assembly** component window, and assembly instructions at addresses 0x800,16 are highlighted**.**

Equivalent operations:

`ATTRIBUTES ADR`	Select menu entry ***Assembly* | *Display Adr***.
`ATTRIBUTES SMEM`	Select a range in memory component window and drag it to the **Assembly** component window.
`ATTRIBUTES SPC`	Drag a register to the **Assembly** component window.
`ATTRIBUTES CODE`	Select menu entry ***Assembly* | *Display Code***.

In the Register Component

Syntax:

`ATTRIBUTES <list>`

Arguments:

`list`=command{,command}

command=`FORMAT(hex|bin)` | `VSCROLLPOS` vposition | `HSCROLLPOS` hposition

`vposition`=*expression*

`hposition`=*expression*

`VSCROLLPOS` vposition = 1
: The second line of registers is on top of the register component.

`VSCROLLPOS` vposition = 0
: Returns to the default display. The first line of registers is on top of the register component.

`HSCROLLPOS` hposition = 1
: The second column of registers is on the left hand side of the register component.

`HSCROLLPOS` hposition = 0
: Returns to the default display. The first column of registers is on the left hand side of the register component. The `HSCROLLPOS` command sets the position of the horizontal scroll box (in column: a column is about a tenth of the greatest register or bitfield width).

`hex` — Sets format representation to hexadecimal

`bin` — Sets format representation to binary

Description:

The `ATTRIBUTES` command sets the display and state options of the **Register** component window. The `FORMAT` command sets the display format of register values. The `VSCROLLPOS` command sets the position of the vertical scroll box.

The attribute `VSCROLLPOS` enables vertical scrolling in the register component. The expression specified is an absolute and positive value for scrolling. This command is used when a vertical scroll bar is present at the right of the register component.

The attribute `HSCROLLPOS` enables horizontal scrolling in the register component. The expression specified is an absolute and positive value for scrolling. This command is used when a horizontal scroll bar is present at the bottom of the register component.

Example:

```
    Register < ATTRIBUTES FORMAT BIN
```

Contents of registers are displayed in binary format in the **Register** component window.

```
    Register < ATTRIBUTES VSCROLLPOS 3
```

Scrolls three positions down. The fourth line of registers is displayed at the top of the register component.

```
    Register < ATTRIBUTES VSCROLLPOS 0
```

Returns to the default display. The first line of registers is displayed at the top of the register component.

```
    Register < ATTRIBUTES HSCROLLPOS 5
```

Scrolls five positions right. The sixth column of registers is displayed at the left of the register component.

```
    Register < ATTRIBUTES HSCROLLPOS 0
```

Returns to the default display. The first column of registers is displayed to the left of the register component.

Equivalent operations:

`ATTRIBUTES FORMAT`	Select menu entry **Register | Options**.
`ATTRIBUTES VSCROLLPOS`	Scroll vertically in the **Register** component window.
`ATTRIBUTES HSCROLLPOS`	Scroll horizontally in the **Register** component window.

In the Source Component

Syntax:

```
ATTRIBUTES <list>
```

Arguments:

`list=command{,command}`

command= `SPC address` | `SMEM` range | `SMOD` module | `SPROC` numberAssociatedToProcedure | `MARKS (ON|OFF)`

`address`	Address to be located
`range`	Memory range to be located
`module`	Specified module
`MARKS ON`	Displays breakpoint marks
`MARKS OFF`	Hides breakpoint marks

Description:

The `ATTRIBUTES` command sets the display and state options of the **Source** component window.

The `SMEM` (show memory range) command and `SPC` (show PC address) command displays the corresponding module’s source text, scrolls to the corresponding text range location or text address location, and highlights the corresponding statements.

The `SMOD` (show module) command displays the corresponding module’s source text. If the module is not found, a message is displayed in the **Command Line** component window. The `SPROC` (show procedure) command loads the corresponding module’s source text, scrolls to the corresponding procedure, and selects the statement that is in the procedure chain of this procedure.

The `SPROC` command is applicable only for C source-level debugging. The `numberAssociatedToProcedure` is the level of the procedure in the procedure chain.

The `MARKS` command `ON` or `OFF` displays or hides the breakpoint marks. Marks are visible in the **Source** component window.

Example:

```
Source < ATTRIBUTES MARKS ON
```

Equivalent operations:

`ATTRIBUTES SPC`	Drag and drop from register component to source component.
`ATTRIBUTES SMEM`	Drag and drop from memory component to source component.
`ATTRIBUTES SMOD`	Drag and drop from module component to source component.
`ATTRIBUTES SPROC`	Drag and drop from procedure component to source component.
`ATTRIBUTES MARKS`	Select menu entry **Source | Marks**.

In the Data Component

Syntax:

```
ATTRIBUTES <list>
```

Arguments:

`list=command{,command}`

command=`FORMAT(bin |oct |hex |signed |unsigned | symb) | MODE(automatic |periodical |locked | frozen)| SMOD` module | `UPDATERATE` rate

`hex`	Sets format representation to hexadecimal
`oct`	Sets format representation to octal
`bin`	Sets format representation to binary
`symb`	Sets format representation as a symbol
`signed`	Displays value in signed decimal format
`unsigned`	Displays value in unsigned decimal format
`periodical`	Sets data component to periodical update mode
`locked`	Sets data component to locked update mode
`frozen`	Sets data component to frozen update mode
`automatic`	Sets data component to automatic update mode
`module`	Specified module
`rate`	Update rate in tenth of a second. Valid value for the rate is 0 .. 600.

Description:

The `ATTRIBUTES` command sets the display and state options of the **Data** component window.

The `FORMAT` command selects the representation used for the list of variables. The representation is one of the following: binary, octal, hexadecimal, signed decimal, unsigned decimal, or symbolic.

The `MODE` command selects the display mode of variables.

In automatic mode (default), variables are updated when the target is stopped. Variables from the currently executed module or procedure are displayed in the data component.

In locked and frozen mode, variables from a specific module are displayed in the data component. In that case, the same variables are always displayed in the data component.

In locked mode, values from variables displayed in the data component are updated when the target is stopped.

In frozen mode, values from variables displayed in the data component are not updated when the target is stopped.

In periodical mode, variables are updated at regular time intervals when the target is running. The default update rate is 1 second, but it can be modified by steps of 100 ms using the associated dialog box or the `UPDATERATE` command.

The `UPDATERATE` command sets the update rate for the data component. This command is only relevant when the update mode for the data component is set to periodical (refer to `UPDATERATE` command).

The `SMOD` (show module) command displays global variables of the corresponding module.

Equivalent operations:

`ATTRIBUTES FORMAT`	Select menu entry **Data | Format...**
`ATTRIBUTES MODE`	Select menu entry **Data | Mode...**
`ATTRIBUTES SMOD`	Drag and drop from module component to data component.
`ATTRIBUTES UPDATERATE`	Select menu entry **Data | Mode | Periodical**.

Example:

```
Data < ATTRIBUTES MODE FROZEN
```

In the data component, the frozen mode is set for global variables. Variables are not refreshed when the application is halted.

In the Memory Component

Syntax:

```
ATTRIBUTES <list>
```

Arguments:

```
list=command{,command}

command=FORMAT(bin|oct|hex|signed|unsigned)|
        WORD number | ADR (ON|OFF)| ASC (ON|OFF)|
        ADDRESS address | SPC address | SMEM range |
        SMOD module | MODE (Automatic |
        Periodical | Frozen) | UPDATRATE rate
```

`hex`	Sets format representation to hexadecimal
`oct`	Sets format representation to octal
`bin`	Sets format representation to binary
`signed`	Displays value in signed decimal format
`unsigned`	Displays value in unsigned decimal format
`number`	Requested word size. The word size can be 1, 2, or 4 bytes.
`address`	Memory address to be located
`range`	Memory range to be located
`module`	Specified module
`periodical`	Set component to periodical update mode
`frozen`	Set component to frozen update mode
`automatic`	Set component to automatic update mode
`rate`	Update rate in tenth of a second. Valid value for the rate is 0 .. 600.

Description:

The `ATTRIBUTES` command sets the display and state options of the **Memory** component window.

The `FORMAT` command selects the display format in the **Memory** window. Format can be set to binary, octal, hexadecimal, signed decimal, unsigned decimal, or symbolic.

The `WORD` command selects the word size of the memory dump window. The word size can be 1, 2, or 4 bytes.

The `ADR` command `ON` or `OFF` displays or hides the address in front of the memory dump lines.

The `ASC` command `ON` or `OFF` displays or hides the ASCII dump at the end of the memory dump lines.

The `ADDRESS` command scrolls the **Memory** component window to the specified address and displays the corresponding memory address (memory `WORD` is not selected).

The `SPC` (Show PC) and `SMEM` (Show Memory) commands scroll the memory component window to the specified address or range of memory.

The `SMOD` (Show Module) command scrolls the memory component window to the address of the first global variable in the specified module.

The `MODE` command selects the display mode for the memory component.

In automatic mode (default mode), the memory component is updated when the target is stopped.

In frozen mode, memory dump displayed in the memory component is not updated when the target is stopped.

In periodical mode, the content of the memory component is updated at regular time intervals when the target is running. The default update rate is 1 second, but it can be modified by steps of 100 ms using the associated dialog box or the `UPDATERATE` command.

The `UPDATERATE` command sets the update rate for the memory component. This command is only relevant when the update mode is set to periodical.

Equivalent operations:

`ATTRIBUTES FORMAT`	Select menu entry **Memory | Format**.
`ATTRIBUTES WORD`	Select menu entry **Memory | Word Size**.
`WORD 1`	The content is displayed in byte format.
`WORD 2`	The content is displayed in word (2-byte) format.
`WORD 4`	The content is displayed in long (4-byte) format.
`ATTRIBUTES ADR`	elect menu entry ***Memory* | *Display* | *Address***.
`ATTRIBUTES ASC`	Select menu entry ***Memory* | *Display* | *ASCII***.

`ATTRIBUTES ADDRESS`	Select menu entry **Memory | Address...**
`ATTRIBUTES SMEM`	Drag and drop from data component (variable) to memory component.
`ATTRIBUTES SMOD`	Drag and drop from source component to memory component.

Example:

```
Memory < ATTRIBUTES ASC OFF, ADR OFF
```

ASCII dump and addresses are removed from the **Memory** component window.

AUTOSIZE

Short description:

Autosizes the component windows in the main window layout

Syntax:

```
AUTOSIZE <on>|<off>
```

Arguments:

`on`	Enables autosizing of windows
`off`	Disables autosizing

Description:

`AUTOSIZE` enables/disables windows autosizing.

When enabled `(on)`, the size of component windows is automatically adapted to the MCUez main window when it is resized.

Example:

```
AUTOSIZE on
```

Windows autosizing is enabled.

BCKCOLOR

Short description:

Sets the background color

Syntax:

```
BCKCOLOR <color>
```

Argument:

`color` Available colors: `BLACK, GREY, LIGHTGREY, WHITE, RED, YELLOW, BLUE, CYAN, GREEN, PURPLE, LIGHTRED, LIGHTYELLOW, LIGHTBLUE, LIGHTCYAN, LIGHTGREEN, LIGHTPURPLE`

Description:

`BCKCOLOR` sets the background color. Ensure that the text will always be visible by using different colors for the font and the background. Do not use colors that have a specific meaning in the **Command Line** window. These colors are:

- Red — To display error messages
- Blue — To echo commands
- Green — To display asynchronous events

When `WHITE` is specified as an argument, the default setting is set for the background of all component windows; for instance, the register component is `LIGHTGREY`.

Example:

```
BCKCOLOR LIGHTCYAN
```

In this example, the background color for all currently opened windows is set to `LIGHTCYAN`. Execute `BCKCOLOR WHITE` to return to the original display.

CLOSE

Short description:

Closes a component

Syntax:

```
CLOSE <component> | *
```

Arguments:

`component`	Component window
`*`	Closes all component windows

Description:

`CLOSE` closes a component window or all windows.

Example:

```
    CLOSE Memory
```

The **Memory** component window will be closed.

FILL

Short description:

Fills a memory range with a value

Syntax:

```
FILL <range> <value>
```

Arguments:

`range`	Address range
`value`	Single byte value

Description:

In the memory component, the `FILL` command fills a corresponding range with the defined value. The `value` must be a single byte pattern (higher bytes ignored).

Equivalent operation:

The **Fill Memory** dialog is available from the **Memory** component pop-up menu or from the **Memory | Fill...** menu entry.

Example:

```
    in>FILL 0x8000..0x8008 0xFF
```

The memory range `0x8000..0x8008` is filled with the value `0xFF`.

FIND

Short description:

Finds and highlights a pattern

Syntax:

```
FIND “<string>” [;B] [;MC] [;WW]
```

Arguments:

`string`	Pattern to match
`;B`	Search backward, default is forward
`;MC`	Match case sensitive
`;WW`	Match whole word

Description:

In the source component, the `FIND` command is used to search for a specified pattern in the source file currently loaded. The search is forward (default), backward (`;B`), match case (`;MC`) or match whole word (`;WW`). The operation starts at the currently highlighted statement or beginning of file (if nothing is highlighted). If the pattern is found, the **Source** window is scrolled to the item and highlighted.

Equivalent operation:

Select **Source | Find...** or open the **Source** component pop-up menu and select **Find...** to open the **Find** dialog.

Example:

```
   in>FIND “thing” ;B ;WW
```

The string `“thing”` is searched in the **Source** component window. The search is performed backward and for a complete word, not part of a word.

FONT

Short description:

Sets the text font

Syntax:

```
FONT ‘FontName’ [size][color]
```

Arguments:

`FontName`	Name of a valid font installed on the system. If the specified font is not found, the operating system will try to find the available font that best fits the specification.
`size`	Defines the font size to use. This parameter must be a positive numerical constant representing a point size.
`color`	Defines the color to use. This parameter can be one of the following: `BLACK, GREY, LIGHTGREY, WHITE, RED, YELLOW, BLUE, CYAN, GREEN, PURPLE, LIGHTRED, LIGHTYELLOW, LIGHTBLUE, LIGHTCYAN, LIGHTGREEN, LIGHTPURPLE`. The color specified in the `FONT` command will be used to display text in all component windows. Do not use the same color for the font and background; otherwise, the content of the component window will not be visible.

Description:

The `FONT` command enables the font type, color, and size to be changed.

Font Color Exceptions — The only exceptions apply to the command line component and source component. The color used in the command line component is fixed and cannot be changed. The prompt and response are always displayed in black, the commands in blue, and error messages in red. The color used in the source component for chroma encoding is also fixed. ANSI C keywords are displayed in blue, comments in green, and strings in red. The rest of the code is displayed using the color specified in the `FONT` command.

Example:

```
in>FONT ‘Arial’ 8 BLUE Text is written in blue
using Arial, 8 point font.
```

Equivalent operation:

The **Font** dialog is available by selecting the **Component | Fonts...** menu entry.

HELP

Short description:

Displays a list of primary commands

Syntax:

`HELP`

Description:

In the command line component, the `HELP` command displays all available primary commands. Subcommands from the `ATTRIBUTES` command are not listed. Component specific commands for components that are not opened will not be listed.

Example:

```
in>help
 MCUez:
   VER         Shows the version of all loaded
               commands
   AUTOSIZE    Selects window sizing mode
   OPEN        Opens a component window
   SET         Loads a target component
 ...
 ...
```

LOAD

Short description:

Loads an application

Syntax:

```
LOAD [applicationName]
```

Argument:

`applicationName`	Name of an application. If the *.abs* file is not located in the project directory, the complete path must be specified in front of the filename.

Description:

`LOAD` loads an application (*.abs* file) for a debugging session. If no parameter is specified, the **Load Executable File** dialog is opened. This error is displayed when no target is installed: Error: No target is installed. This error message is displayed when no target is connected: Error: No target is connected.

Example:

```
in>LOAD FIBO.ABS
```

Loads the application *fibo.abs*

OPEN

Short description:

Opens a component window

Syntax:

```
OPEN <componentName> [x y width height ][;I]
```

Arguments:

`componentName`	Name of component window to be opened
`x`	X axis of the component window upper-left corner
`y`	Y axis of the component window upper-left corner
`width`	Component window width
`height`	Component window length
`I`	Reduces the component window to an icon

`x, y, width,` and `height` are specified in percentages of the main window.

Description:

`OPEN` opens a component. If `I` is set, the component window is iconized.

Example:

```
in>OPEN Terminal 0 78 60 22
```

The terminal component and corresponding window are opened at specified positions and with specified width and height.

SLAY

Short description:

Saves the general window layout

Syntax:

```
SLAY <filename>
```

Argument:

`filename`	Name of file (with full path) where the layout is saved

Description:

The `SLAY` command is used to save the format and layout of all component windows in a *.hwl* file.

Example:

```
   in> slay /mcuez/demo/mylayout.hwl
```

The current debugger layout is saved in the file *mylayout.hwl*.

Layout files usually have an *.hwl* extension. However, any file extension can be specified. If no path is specified, the destination directory is the current project directory.

SMEM

Short description:

Shows a memory range

Syntax:

```
SMEM <range>
```

Argument:

`range`	Address range

Description:

This command applies to the **Source**, **Assembly**, and **Memory** component windows.

In the source component, the `SMEM` command displays the corresponding module's source text, scrolls to the corresponding text location (the code address), and highlights the statements which correspond to this code address range.

In the assembly component, the `SMEM` command scrolls the assembly component, shows the location (the assembler address), and select/highlights the memory lines of the address range given as an argument.

In the memory component, the `SMEM` command scrolls the memory dump component, shows the locations (memory address) of the address range given as the argument.

Example:

```
in>Source < SMEM 0x8000..0x8008
```

The **Source** component window scrolls to the source code corresponding to the instruction located at address 0x8000. The source code generating code between address `0x8000` and `0x8008` is highlighted.

```
in>Memory < SMEM 0x8000,8
```

The **Memory** component window scrolls to the address 0x8000 and the memory range `0x8000..0x8007` is highlighted.

```
in>SMEM 0x8000..0x8008
```

Without redirection, this command applys to all source, assembly, and memory components.

SMOD

Short description:

Shows a module

Syntax:

```
SMOD <module>
```

Argument:

`module`	Name of a module bound to the application. The module name should contain no path. The module extension (*.dbg* for assembly sources or *.c* for C sources) must be specified.

Description:

This command can be redirected to the **Source**, **Assembly**, and **Memory** component windows. Without redirection, this command applys to all three component windows.

In the source component, the `SMOD` command displays the corresponding module's source text. If the module is not found, a message is displayed in the **Command Line** window.

In the data component, the `SMOD` command loads the corresponding module global variables.

In the memory component, the `SMOD` command scrolls the memory dump component and highlights the first global variable of the module.

The module is searched for in the directories specified in the `GENPATH` environment variable. An error message is displayed:

- If the specified module is not bound to the application currently loaded
- If no application is loaded

Example:

```
in>Data < SMOD fibo.dbg
```

Global variables defined in the *fibo.dbg* module are displayed in the data component window.

The `module` argument must be a module filename given in the **Module** component window.

SPC

Short description:

Shows the specified address in a component window

Syntax:

```
SPC <address>
```

Argument:

`address`	User-specified address

Description:

This command can be redirected to the **Source**, **Assembly**, and **Memory** component windows. Without redirection, this command applys to all three component windows.

In the source component, the `SPC` command loads the corresponding module’s source text, scrolls to the corresponding text location (the code address), and highlights the statement that corresponds to this code address.

In the assembler component, the `SPC` command scrolls the assembly component, shows the location (assembler address), and selects/highlights the assembler instruction of the address given as the parameter.

In the memory component, the `SPC` command scrolls the memory dump component and shows the location (memory address) of the address given as the parameter.

Example:

```
   in>Assembly < SPC 0x8000
```

The **Assembly** component window scrolls to the address 0x8000 and the instruction located there is highlighted.

SPROC

Short description:

Shows information associated with the specified procedure. This command is available only when performing C source-level debugging using the MCUez compiler. Refer to **Appendix B. C Source-Level Debugging** for information on C source debugging.

Syntax:

```
SPROC <level>
```

Argument:

`level`	Stack level

Description:

In the data component, the `SPROC` command shows local variables of the corresponding procedure stack level.

In the source component, the `SPROC` command loads the corresponding module's source text, scrolls to the corresponding procedure and highlights the statement of this procedure that is in the procedure chain.

Example:

```
    in>Data < SPROC 0
```

UPDATERATE

Short description:

Sets the data update mode

Syntax:

```
UPDATERATE <rate>
```

Argument:

`rate`	Constant number representing time in tenths of a second (1 – 600 = 0.1 to 60 seconds)

Description:

In the data and memory components, the `UPDATERATE` command is used to set the data refresh update rate. The `UPDATERATE` command is in effect only when the data or memory component is set to periodical mode.

Example:

```
    in>Memory <updaterate 30
```

Sets memory to update every three seconds

VER

Short description:

Displays the version number

Syntax:

```
VER
```

Description:

The VER command displays the MCUez version number and currently loaded components in the **Command Line** component window.

Example:

```
in>ver
MCUez                2.0.26
MCUez Engine         2.0.48
Source               2.0.19
Assembly             2.0.13
Register             2.0.13
Memory               2.0.18
Data                 2.0.26
Command Line         2.0.15
Module               2.0.4
ElfLoader            2.0.16
```

ZOOM

Short description:

Zooms in/out on a variable

Syntax:

```
ZOOM (address in| [address] out)
```

Argument:

`address`	The address of the structure or pointer variable that should be zoomed-in or zoomed-out

Description:

In the data component, the `ZOOM in` command is used to display the member fields of structures. Member fields are not expanded in place. The member fields display replaces the previous view. The `ZOOM out` command is used to return to the nesting level indicated by the given identifier. Addresses are not needed to zoom out. Simply type `ZOOM out`.

Example:

```
in>ZOOM 0x1FE0 in
```

The variable structure located at address `0x1FE0` is zoomed in.

This command is relevant when C source debugging.

```
in>zoom &_StartupData
```

The previous example zooms in on the `_StartupData` structure and displays member fields and values.

5.10 Command Files

The command files *startup.cmd*, *reset.cmd*, *preload.cmd*, and *postload.cmd* are MCUez system command files. These command files do not exist automatically. They could be installed when installing a new target. However, the debugger is able to recognize these command files and execute them.

- *startup.cmd* is executed when a target interface is loaded (the target defined in the *project.ini* file or when **Component | Set Target** in the menu is selected).
- *reset.cmd* is executed when the **Target Name | Reset** menu entry is selected **Target Name** is the name of the target, such as D-Bug12, SDI, etc..
- *preload.cmd* is executed before loading an *.abs* application file (when **Target Name | Load...** is selected.
- *postload.cmd* is executed after loading an *.abs* application file.

Section 6. D-Bug12 Monitor Target Component

6.1 Contents

6.2 Introduction

This section describes the D-Bug12 monitor target component as it relates to the MCUez debugger environment.

The D-Bug12 monitor target component is an interface used to communicate with Motorola's M68EVB912B32 and M68EVB812A4 evaluation boards.

6.3 General Description

The MCUez D-Bug12 monitor target component establishes the connection to the D-Bug12 monitor. The D-Bug12 monitor is the program code that resides in the FLASH of the MCU chip on the evaluation board. The MCUez debugger GUI (graphical user interfacer) functions and **Command Line** component window provide the user interface necessary to submit commands to the D-Bug12 monitor.

Commands entered on the command line or selected from the D-Bug12 menu options are translated into D-Bug12 monitor commands and sent to the D-Bug12 monitor code on the evaluation board. The D-Bug12 monitor code processes commands received from the MCUez debugger. Results are sent back to the MCUez debugger and displayed in the appropriate component windows, such as the **Memory**, **Register**, or **Command Line** component windows. **Figure 6-1** shows a general setup between the MCUez debugger software running on the host computer and the evaluation boards.

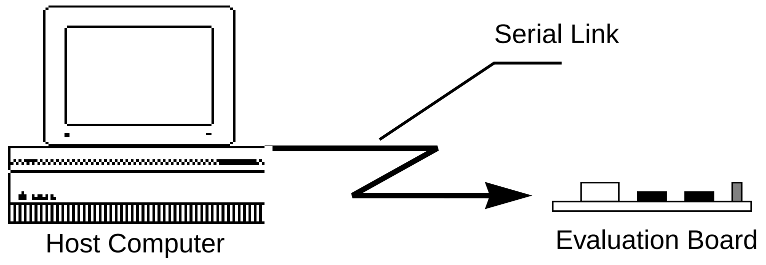


Figure 6-1. General Setup

The MCUez debugger will control and monitor the MCU on the evaluation board. The MCUez debugger can read and write in internal/external memory, single-step/run/stop the CPU, and set breakpoints in the code.

Memory can be accessed while the CPU is running, by stopping the application, accessing memory, and then resume execution.

NOTE: *MCUez for the D-Bug12 supports only EVB mode for all HC12 EVB boards.*

6.4 Interfacing Host Computer and Evaluation Board

6.4.1 Evaluation Board Configuration

Evaluation boards must be properly configured to activate the D-Bug12 monitor. Refer to the appropriate manual, such as the *M68EVB912B32 Evaluation Board User's Manual*, Motorola document order number 68EVB912B32UM/D, or *M68HC12A4EVB Evaluation Board User's Manua*l, Motorola document order number HC12A4EVBUM/D, for proper configuration.

6.4.2 Hardware Connection

The M68EVB912B32 or M68EVB812A4 evaluation board must be connected to the host computer COM port with a standard serial communication cable. The cable is connected to the P1 port on the M68EVB912B32 or J3 port on the M68EVB812A4. Press the reset button on the evaluation board to prepare the EVB board for a connection.

6.5 Loading the D-Bug12 Target Component

If D-Bug12 is not defined as the target component in the *project.ini* file, load the D-Bug12 monitor target component by selecting **Component | Set Target...** as shown in **Figure 6-2** from the list box to load the D-Bug12 target component. The target is set in the *project.ini* file, for example `Target=D-Bug12`.

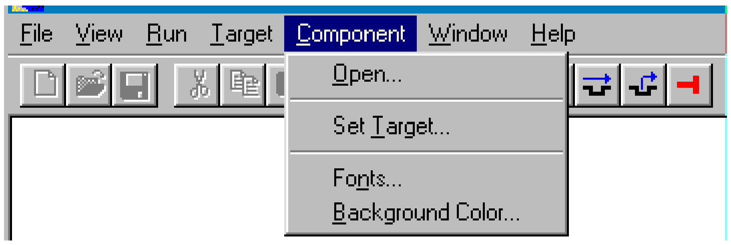
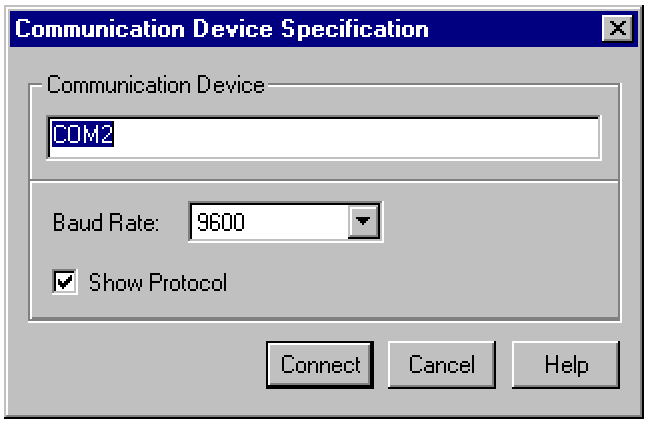


Figure 6-2. Loading D-Bug12

The D-Bug12 monitor target component automatically detects that the evaluation board is connected to the host computer. If the board is not detected, the **Communication Device** dialog box (**Figure 6-3**) pops up, indicating that the board is not connected, is attached to a different port, or the jumpers on the evaluation board are not set correctly.



COM2

Baud Rate: 9600

☑ Show Protocol

Figure 6-3. Communication Device Dialog Box

Enter an available communication device (port) in the **Communication Device** edit box, select a baud rate and click **Connect**. If a connection cannot be established for the selected baud rate, the debugger automatically tries 57,600, 38,400, 28,800, 19,200, 9600, ... 1200. The specified communication device (for example COM2) is saved in the *project.ini* file. The default device is COM1 and the default speed is 9600 baud.

If **Show Protocol** is checked, all commands and responses sent and received are displayed in the **Command Line** window. This feature is used by Motorola personnel for diagnostic purposes.

6.6 Startup Command File

The startup command file (*startup.cmd*) is executed by the MCUez debugger after the D-Bug12 target component has been loaded. This file must be located in the working directory. The user can put any MCUez debugger command in this file to set up the evaluation board hardware before loading an application.

Example of a *startup.cmd* file:

```
wb 0x0035  0x00
wb 0x0012   0x11
baudrate    19200
protocol    off
```

6.7 D-Bug12 Menu Entries

After loading the D-Bug12 monitor target component, the **Target** menu is replaced by the **D-Bug12** menu as shown in **Figure 6-4**.

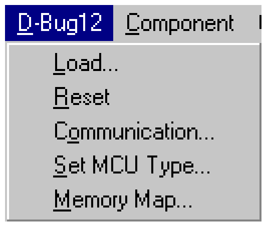


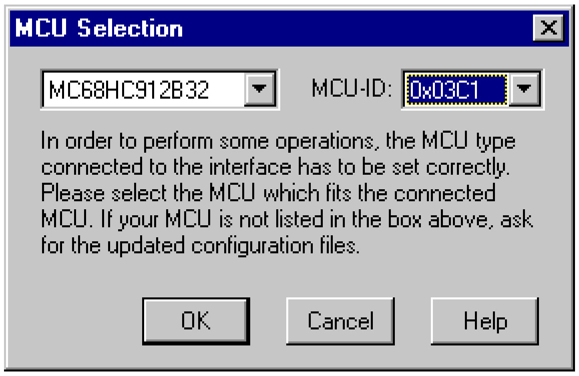
Figure 6-4. D-Bug12 Menu

Select **D-Bug12 | Load...** to load the application to be debugged (an *.abs* file).

D-Bug12 | Reset is not supported for EVB mode.

Select **D-Bug12 | Communication...** to display the **Communication Device** dialog box. If the target is not connected, the **Communication...** menu entry is replaced with the **Connect** menu entry. Select **Connect** to display the **Communication Device** dialog box and re-establish the connection.

Select **D-Bug12 | Set MCU Type...** to open the dialog box shown in **Figure 6-5**.



In order to perform some operations, the MCU type connected to the interface has to be set correctly. Please select the MCU which fits the connected MCU. If your MCU is not listed in the box above, ask for the updated configuration files.

Figure 6-5. MCU Selection Dialog Box

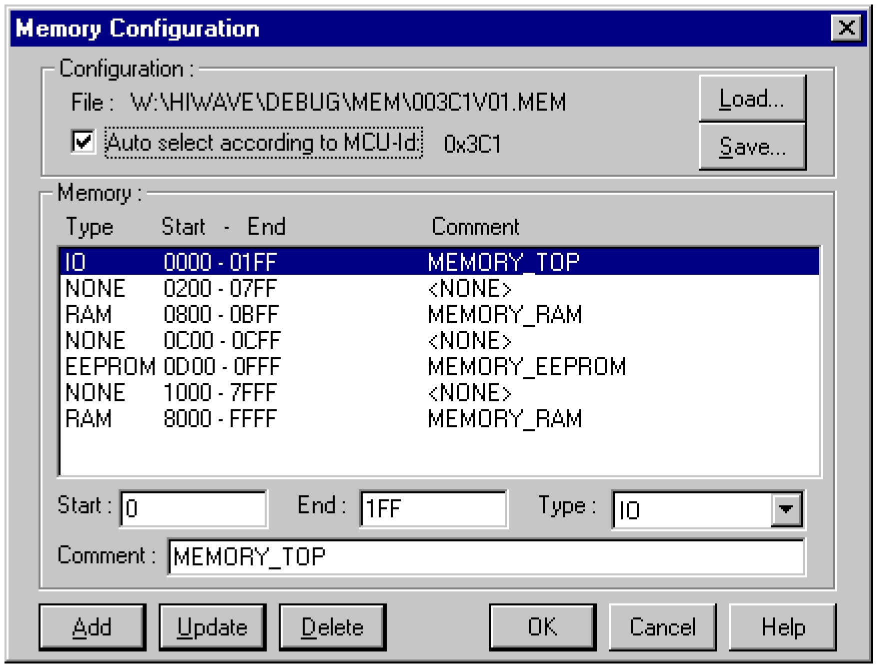
This dialog box allows the user to select the MCU. The currently selected MCU and MCU Id are shown in the dialog box.

Information is taken from the file *mdsemcu.ini*. If a specific MCU is not found in this file, the user is advised to update their installation.

The selection will be saved and used as the default for the next session.

CAUTION: *Due to a design limitation on the EVB boards, if an incorrect MCU or MCU Id is selected for the evaluation board, the connection will be established but the memory map will be misrepresented and writing to the EEPROM may fail. The user must verify that the MCU selected matches the MCU on the evaluation board.*

Select **D-Bug12 | Memory Map...** to open the dialog box shown in **Figure 6-6**.



Configuration :
File : W:\HIWAVE\DEBUG\MEM\003C1V01.MEM
[✓] Auto select according to MCU-Id: 0x3C1

Memory :

Type	Start - End	Comment
IO	0000 - 01FF	MEMORY_TOP
NONE	0200 - 07FF	<NONE>
RAM	0800 - 0BFF	MEMORY_RAM
NONE	0C00 - 0CFF	<NONE>
EEPROM	0D00 - 0FFF	MEMORY_EEPROM
NONE	1000 - 7FFF	<NONE>
RAM	8000 - FFFF	MEMORY_RAM

Start : 0 End : 1FF Type : IO
Comment : MEMORY_TOP

Figure 6-6. Memory Configuration Dialog Box

The **Memory Configuration** dialog displays the default memory layout for the configured MCU. This is for memory map display only. Any changes will not affect memory configuration.

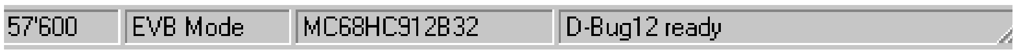
Information about the memory layout is read from the MCU-specific personality file. The personality filename is constructed like this:

00nnnVvv.MEM

where *nnn* is the hexadecimal representation of the MCU Id (three digits) and *vv* is the version number. This file is searched for in the *PROG\MEM* subdirectory.

6.8 Status Bar

Information about the D-Bug12 monitor target component is displayed in the debugger status bar. The baud rate, current evaluation board mode, the MCU and the current status are displayed from left to right in the status bar.



6.9 D-Bug12 Default Environment

The next example indicates the parameter in the *default.env* file that pertains to the D-Bug12 target.

Example:

```
MCUID=0x3C1
```

If this parameter is specified, the corresponding register file is loaded from the current working directory.

6.10 D-Bug12 Target Component Commands

The following commands can be entered in the command line component or specified in any command script file (such as *startup.cmd*).

6.10.1 PROTOCOL

The `PROTOCOL` command displays the communication protocol between the MCUez debugger and D-Bug12 monitor in the **Command Line** window. This feature is used by Motorola personnel for diagnostic purposes. The default is `OFF`.

Syntax:

```
PROTOCOL < ON | OFF >
```

Example:

```
PROTOCOL ON
```

6.10.2 BAUDRATE

This command is used to change the baud rate from the **Command Line** window. Possible rates are 9600, 19,200, 28,800, 38,400, and 57,600.

Syntax:

```
BAUDRATE < RATE >
```

Example:

```
BAUDRATE 19200
```

6.10.3 PT

With the `PT` (pass through) command, it is possible to use D-Bug12 monitor commands from the **Command Line** component window. This command is intended for use by Motorola personnel only.

6.10.4 VER

The `VER` command displays the version of the D-Bug12 monitor target component, followed by output from the D-Bug12 monitor `device` command.

Syntax:

```
VER
```

Example:

```
....
D-Bug12 Target   5.3.2
Device: 912B32
EEPROM: $0D00 - $0FFF
FLASH: $8000 - $FFFF
RAM: $0800 - $0BFF
I/O Regs: $0000
```

6.10.5 DEVICE

If the user remaps the EEPROM to another location, this command must be used to inform the MCUez debugger and D-Bug12 monitor where the EEPROM now resides. Refer to the *M68EVB912B32 Evaluation Board User's Manual*, Motorola document order number 68EVB912B32UM/D, for parameters.

Syntax:

```
DEVICE <parameters>
```

Example:

```
DEVICE  DG128 800 FFF 4000 FFFF 1000 1bFF 0
```

6.11 Communication Scenario

The MCUez debugger communicates with the M68EVB912B32 evaluation board. The debugger uses commands interpreted by the D-Bug12 monitor to perform specific functions.

Table 6-1 lists the MCUez debugger functions (left column) that correspond with the D-Bug12 monitor commands (right column).

6.11.1 Stop Program in EVB Mode

The D-Bug12 monitor does not support stopping the program. To stop the program, the SCI receiver interrupt is used. Before the application is started, the receiver interrupt enable flag is set by the MCUez debugger. If a character is sent to the target, an SCI0 interrupt occurs and the message SCI0 Exception is sent to the MCUez debugger.

6.11.2 User-Defined Vectors

User-defined interrupts cannot be used because the interrupt jump table is located in FLASH EEPROM (0xF7C0..0xF7CF). Therefore, it is not possible to initialize the vector table.

Table 6-1. MCUez Debugger Functions and Related D-Bug12 Monitor Commands

Function	D-Bug12 Command
read memory	upload <startadr> <endadr>
write memory	load <adr>
read register	rd
write register	<register> <value>
read PC	rd
write PC	pc <value>
set breakpoint	br <adr> ...
delete breakpoint	nobr <adr> ...
start program	g
single step	t
halt program	EVB mode: see **6.11.1 Stop Program in EVB Mode**
reset	EVB mode: Not supported by software; can be manually reset by S1 reset button on EVB

6.12 M68EVB912B32 Evaluation Board

The D-Bug12 monitor supports several operating modes of the evaluation board. However, MCUez for D-Bug12 only supports EVB mode for the M68EVB912B32.

In EVB mode, the application loaded into RAM on the evaluation board is executed and controlled through the MCUez debugger.

6.12.1 Operating Modes

The D-Bug12 monitor is an on-chip (HC912B32) FLASH EEPROM program located in $8000–$F67F. It is started automatically when the board is powered-up or reset in EVB mode.

If the D-Bug12 monitor is accidentally removed or overwritten, the MCUez debugger will no longer monitor the board.

NOTE: *From a dumb terminal, the user can reprogram the D-Bug12 monitor through the bootload mode (refer to Appendix E in the M68EVB912B32 Evaluation Board User's Manual, Motorola document order number 68EVB912B32UM/D).*

6.12.2 Memory Map

In EVB mode, the application must be loaded into the memory area 0x800..0x9FF. The memory area 0xA00..0xBFF is used by the D-Bug12 monitor program and must not be used.

It is not possible to define user vectors because the interrupt vector jump table is located in the FLASH EEPROM area.

Table 6-2. M68EVB912B32 Memory Map

Address Range	Usage	Description
$0000–01FF	CPU registers	On-chip registers
$0800–$09FF $0A00–$0BFF	User code/data Reserved for D-Bug12	1-K on-chip RAM
$0D00–$0FFF	user code/data	768 bytes on-chip EEPROM
$8000–$F67F $F680–$F6BF $F6C0–$F6FF $F700–$F77F $F780–$F7FF $F800–$FBFF $FC00–$FFBF $FFC0–$FFFF	D-Bug12 code User-accessible functions D-Bug12 customization data D-Bug12 startup code Interrupt vector jump table Reserved for bootloader expansion EEPROM bootloader Reset and interrupt vectors	32-Kbytes on-chip FLASH EEPROM

6.13 M68HC12A4EVB Evaluation Board

Table 6-3. M68HC12A4EVB Memory Map

Address Range	Description	Location
\$0000–\$01FF	CPU registers	On-chip (MCU)
\$0800–\$09FF \$0A00–\$0BFF	User code/data Reserved for D-Bug12	1-K on-chip RAM (MCU)
\$1000–\$1FFF	User code/data	4-K on-chip EEPROM (MCU)
\$4000–\$7FFF	User code/data	16-K external RAM (U4, U6A)
\$8000–\$9FFF \$A000–\$FD7F \$FD80–\$FDFF \$FE00–\$FE7F \$FE80–\$FEFF \$FF00–\$FF7F \$FF80–\$FFFF	Available for user programs* D-Bug12 program D-Bug12 startup code* User-accessible functions D-Bug12 customization data* Available for user programs* Reserved for interrupt and reset vectors	32-K external EPROM (U7, U9A)

* Code in these areas may be modified. Requires reprogramming of the EPROMs; refer to Appendix E Customizing the EPROMs in the *M68HC12A4EVB Evaluation Board User's Manual*.

Ensure that jumper settings are default settings (D-Bug12) and not SDI settings (background debug mode).

For more information, refer to the *M68HC12A4EVB Evaluation Board User's Manual*, Motorola document order number HC12A4EVBUM/D.

Section 7. FLASH Programming

7.1 Contents

7.2 Introduction

The non-volatile memory control (NVMC) component specified in this section is an extension for the MCUez debugger and allows the user to control on-chip FLASH devices.

The FLASH component is designed to be flexible and support a large number of MCUs and FLASH modules. This is achieved by the MCUez debugger, which loads an MCU parameter file (*.fpp*). The parameter file provides all MCU details (structure, access algorithms, location).

Special algorithms are used to write into the MCU FLASH, EEPROM, or other non-volatile memory modules. FLASH devices have to be erased before they can be written and may need some initialization to be accessible.

The NVMC component lists all non-volatile memory modules and associated details (structure, state, and location). The user can change the state (enabled/disabled, blank, programmed, protected, unprotected, arm, or disarm) and program data into these modules.

7.3 NVMC Graphical User Interface

The NVMC component is integrated with the MCUez debugger as an extension of the SDI target component. If the NVMC component is available, the **FLASH...** menu entry appears in the **SDI** menu located on the **Debugger** menu bar.

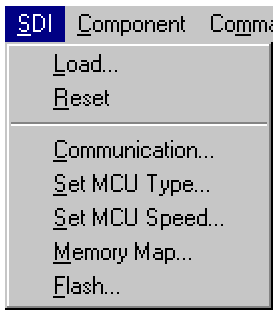
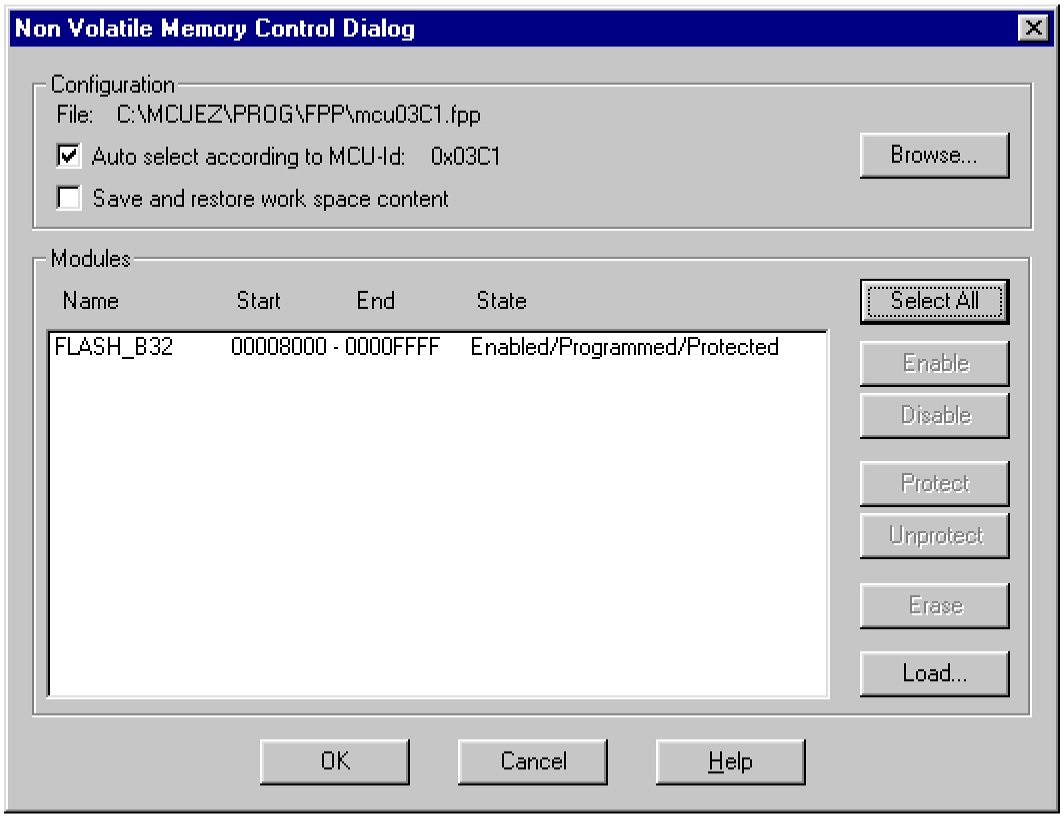


Figure 7-1. SDI Target Menu

Select **FLASH...** from the menu to display the **NVMC** dialog box. This dialog consists of a **Configuration** group and a **Modules** listbox that lists all FLASH or EEPROM blocks. If a module consists of several blocks, each block is listed. Operations performed on a module will affect all blocks in the module.

Each block has a line containing this information:

1. Name of block (according to CPU: B32, A4, etc.)
2. Start address of block
3. End address of block
4. State of the block (disabled, enabled, blank, programmed, protected, or unprotected)


Non Volatile Memory Control Dialog

Configuration
File: C:\MCUEZ\PROG\FPP\mcu03C1.fpp
[✓] Auto select according to MCU-Id: 0x03C1
[ ] Save and restore work space content

Modules

Name	Start	End	State
FLASH_B32	00008000 -	0000FFFF	Enabled/Programmed/Protected

Figure 7-2. NVMC Dialog Box

Available state combinations are:

- Bad device (correct device not detected)
- Disabled
- Enabled/blank/protected
- Enabled/blank/unprotected
- Enabled/programmed/protected
- Enabled/programmed/unprotected

Enabled	A block that is currently active on-chip. It can be read (as ROM) and programmed. For CPUs that consist of blocks that are always active, the state is not displayed in the **State** field of the dialog box.
Disabled	The block is not active. It cannot be programmed and reading it will return invalid values.
Blank	A block is empty. All bytes are 0xFF or 0x00, (depending on hardware) and can be programmed on its full address range.
Programmed	A block is fully or partially programmed (not all bytes are 0xFF or 0x00) and cannot be programmed or can be programmed only in some areas. The user must know which areas are available for programming.
Protected	A block is partially or fully protected from erasing and programming.
Unprotected	Protection is off. This allows the block to be erased or programmed. For CPUs that consist of blocks that are always unprotected, the state is not displayed in the **State** field of the dialog box.

NOTE: *A state is displayed if it is applicable. For example,* **Enabled** *is displayed only if it is possible to* **Disable** *a block and* **Unprotected** *is displayed if a block can be* **Protected***.*

In the **Configuration** group, the name of the current FLASH parameter file (*.fpp*) is shown. Check **Auto select according to MCU-Id** to automatically load the parameter file associated with the MCU.

Check **Save and restore workspace content** (equivalent to commands `SAVECONTEXT` and `LOADCONTEXT`) to save the current RAM data, since FLASH programming applications are loaded into RAM and will overwrite the data. Saving the data will slow down the NMVC.

Blocks can be selected with the mouse. Use the shift or Ctrl key with the left mouse button to select multiple blocks. Use the Ctrl key and left mouse button to unselect a block. Selections made in the **NVMC** dialog box or in the **Command Line** component window with the `FLASH select` or `unselect` commands are equivalent.

7.3.1 Handling FLASH Module

FLASH operations can be executed also from the debugger **Command Line** window. Corresponding commands are described in **7.4 NVMC Commands**.

The **Enable**, **Disable**, **Protect**, **Unprotect**, and **Erase** buttons in the **NVMC** dialog box apply to each block. All buttons are dynamic, which means they are active if the function is possible for at least one of the blocks selected. Otherwise, buttons are disabled.

The **Select All...** button will select all blocks listed in the list box.

The **Enable** button enables all selected blocks that are currently disabled. The **Disable** button disables all selected blocks currently enabled. It is not always possible to enable or disable a FLASH block; this depends on the MCU features and context.

The **Protect** button will protect all selected blocks that are currently unprotected. It is not always possible to protect or unprotect a FLASH block; this depends on the MCU features and context. Protection is provided by a block protection bit internal to the MCU. The **Unprotect** button will unprotect all selected blocks currently protected.

NOTE: *For some MCUs, FLASH blocks are only partially protectable. The boot block and boot routines are protected, not the whole module.*

The **Erase** button is enabled if a selected block is not blank. When a block is erased, the block state will be blank.

The **Load...** button will arm all selected blocks, execute a `LOAD` command, and disarm the blocks again. If no FLASH block is selected, click the **Load...** button to select all blocks to be loaded into FLASH.

7.3.2 FLASH Programming Parameter File

FLASH operations (enable, protect, etc.) are defined by code applets provided in FLASH parameter files. The *.fpp* files contain MCU- dependent parameters and programs to handle internal FLASH modules.

When the **NVMC** dialog box is opened, the *.fpp* file is loaded as follows:

1. The check box **Auto select according to MCU Id** is automatically checked and the file associated with the MCU Id is loaded from the \FPP subdirectory created during installation. If this file is not found, an error message is displayed.
2. If the file is found but has the wrong format, an error message is displayed.
3. In all cases, the MCU Id will be displayed if it is available from the target.

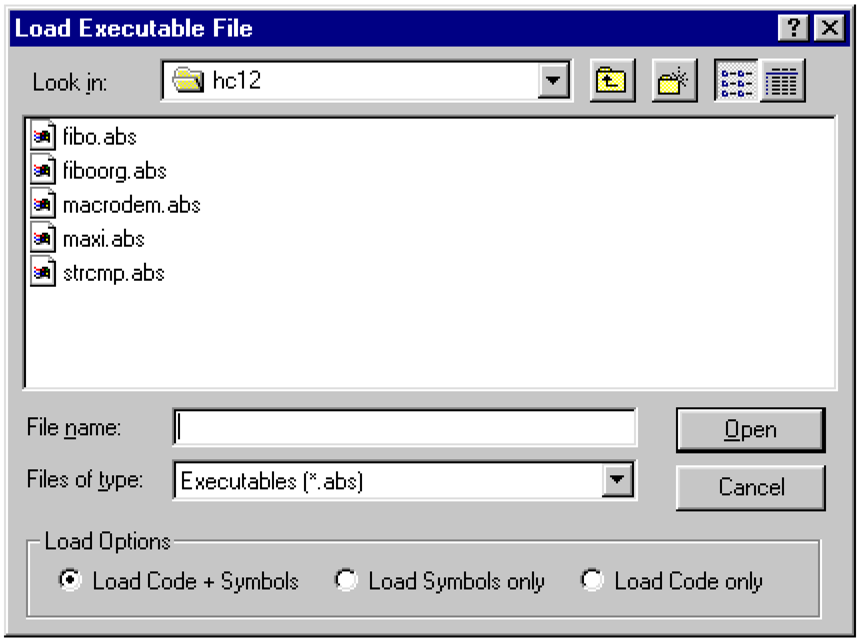
In the **NVMC** dialog box, check **Auto select according to MCU Id*:*** to automatically load the corresponding parameter file.

After the parameter file is loaded, uncheck **Auto select according to MCU Id:** then select the **OK** button. The parameter file will then be assigned to the `NV_PARAMETER_FILE` variable in the *project.ini* file. If the option is checked, the parameter file will not be assigned in the project file.

Select the **Browse...** button to manually find the desired *.fpp* parameter file. If a valid file is loaded, the check box **Auto select according to MCU Id*:*** is automatically unchecked. If an error occurs, an error message is displayed.

7.3.3 Loading an Application in FLASH

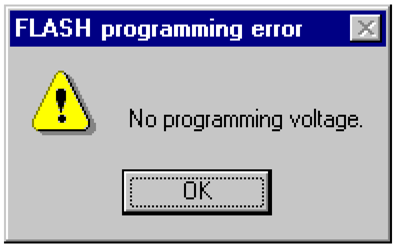
The **Load...** button allows selection of an application file to be loaded (program into FLASH EEPROM). If no block is selected before clicking the **Load** button, then all blocks will be selected. Otherwise, only selected blocks will be affected.



● Load Code + Symbols ○ Load Symbols only ○ Load Code only

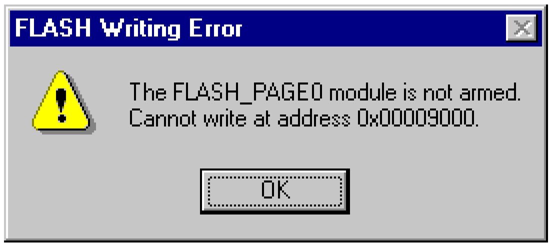
Figure 7-3. Loading Executable File

If a problem occurs while attempting to load an application into FLASH, an error message is displayed. If no programming voltage is available, the following programming error message appears.


FLASH programming error

No programming voltage.

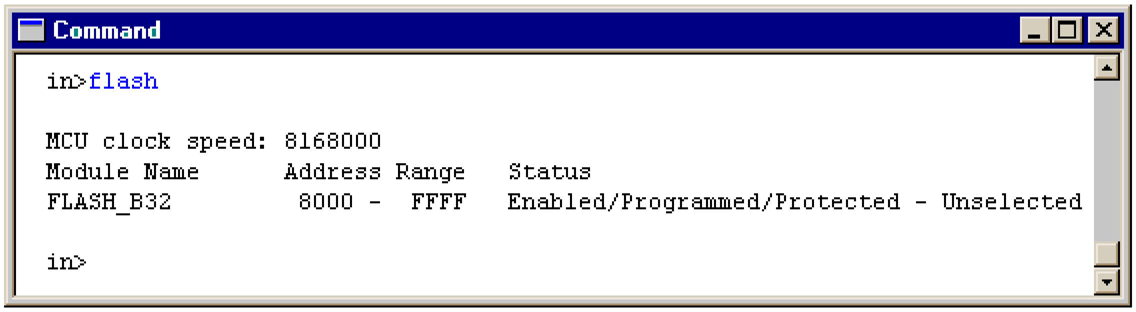
If an attempt is made to load a program into a block that was not selected, an error message will appear indicating which block is not armed.



The FLASH_PAGE0 module is not armed.
Cannot write at address 0x00009000.

7.4 NVMC Commands

Figure 7-4 illustrates FLASH information listed in the **Command Line** window of the debugger.



```
in>flash

MCU clock speed: 8168000
Module Name      Address Range   Status
FLASH_B32         8000 -  FFFF   Enabled/Programmed/Protected - Unselected

in>
```

Figure 7-4. FLASH Commands Display

7.4.1 FLASH

If a FLASH parameter file is loaded, the `FLASH` command performs FLASH operations or displays all available FLASH blocks with their name, location, and state.

If no parameter file is loaded, a file that corresponds to the current MCU Id or the last used *.fpp* file is loaded.

Syntax:

```
FLASH
[(SELECT|UNSELECT|ERASE|ENABLE|DISABLE|PROTECT|
UNPROTECT) [<blockNo>]]
|[ARM|DISARM|SAVECONTEXT|LOADCONTEXT]
|[INIT <fileName>|<mcuId>]
```

`FLASH INIT <fileName>|AUTOID` loads the specified parameter file (including path). If `AUTOID` is specified, the file will be based on the MCU Id.

`FLASH ENABLE` enables the specified blocks. If no parameter is specified, all disabled blocks are enabled. Not all blocks can be disabled and, therefore, cannot be enabled.

`FLASH DISABLE` disables the specified blocks. If no parameter is specified, all enabled blocks are disabled. Not all blocks can be disabled. Disabling of such modules will be ignored.

`FLASH ERASE` erases the specified blocks. If no parameter is specified, all programmed blocks are erased.

`FLASH UNPROTECT` unprotects the specified blocks. If no parameter is specified, all protected blocks are unprotected. Not all blocks can be unprotected. Unprotecting of such blocks will be ignored.

`FLASH PROTECT` protects the specified blocks. If no parameter is specified, all unprotected blocks are protected. Not all blocks can be protected. Protecting of such blocks will be ignored.

`FLASH SELECT` will select specified blocks for FLASH programming. If no parameter is specified, all unselected blocks are selected.

`FLASH UNSELECT` will unselect specified blocks. If no parameter is specified, all selected blocks are unselected. Note that the unselected state is protection against accidental FLASH programming.

`FLASH ARM` prepares the blocks to be loaded by the NMVC with the `LOAD` command. The *vppon.cmd* file is executed. This command is necessary before programming the FLASH.

`FLASH DISARM` ends the programming process. The *vppoff.cmd* file is executed.

`FLASH SAVECONTEXT` saves the current SRAM data into a buffer.

`FLASH LOADCONTEXT` restores the current buffer data into the MCU's SRAM.

`blockNo` is a list of FLASH block/module numbers. Blocks are sequentially numbered, beginning with 0. For example 0, 1, 2, etc. The syntax is:

```
blockNo = {number[-number][,]}
```

Examples:

```
FLASH ERASE 2,7    ;erases memory blocks 2 and 7
FLASH ERASE 2,4-6 8;erases memory blocks 2, 4, 5, 6 and 8
FLASH ERASE        ;erases all available memory blocks
```

The typical sequence to program a FLASH block from the **Command Line** window is:

1. Select the FLASH blocks.
2. Enable the blocks.
3. Unprotect the blocks.
4. Erase the blocks.
5. Arm the blocks.
6. Load (program) the blocks.
7. Disarm the blocks.

NOTE: *In the* **NVMC** *dialog box, the* **Load...** *button automatically arms and disarms the FLASH blocks to program the blocks.*

While FLASH blocks are armed, debugger functions are not possible and will be ignored (running, stepping, etc.). If such a command is entered, a message box will be displayed, asking the user to click **OK** to disarm the blocks and execute the command or to click **CANCEL** to abort the command.

An example of FLASH commands entered on the command line is:

```
in>FLASH INIT mcu03c4.fpp

FLASH parameters loaded for M68HC912DG128 from
C:\MCUEZ\PROG\FPP\mcu03c4.fpp
MCU clock speed: 7982000
Module Addr RangeStatus
FLASH_40004000-7FFFEnabled/Programmed/Protected - Unselected
FLASH_PAGE08000-BFFFEnabled/Programmed/Protected - Unselected
FLASH_C000C000-FFFFEnabled/Programmed/Protected - Unselected
FLASH_PAGE118000-1BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE228000-2BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE338000-3BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE448000-4BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE558000-5BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE668000-6BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE778000-7BFFFEnabled/Programmed/Protected-Unselected
Halted

in>FLASH select 1
in>FLASH enable 1
in>FLASH unprotect 1

Operation passed on FLASH_PAGE0 FLASH module
MCU clock speed: 7978000
Module Addr RangeStatus
FLASH_40004000-7FFFEnabled/Programmed/Protected - Unselected
FLASH_PAGE08000-BFFFEnabled/Programmed/Unprotected - Selected
FLASH_C000C000-FFFFEnabled/Programmed/Protected - Unselected
FLASH_PAGE118000-1BFFFEnabled/Blank/Unprotected - Unselected
FLASH_PAGE228000-2BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE338000-3BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE448000-4BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE558000-5BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE668000-6BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE778000-7BFFFEnabled/Programmed/Protected-Unselected
Halted

in>FLASH erase 1
```

```
Operation passed on FLASH_PAGE0 FLASH module
MCU clock speed: 7982000
Module  Addr RangeStatus
FLASH_40004000-7FFFEnabled/Programmed/Protected - Unselected
FLASH_PAGE08000-BFFFEnabled/Blank/Unprotected - Selected
FLASH_C000C000-FFFFEnabled/Programmed/Protected - Unselected
FLASH_PAGE118000-1BFFFEnabled/Blank/Unprotected - Unselected
FLASH_PAGE228000-2BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE338000-3BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE448000-4BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE558000-5BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE668000-6BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE778000-7BFFFEnabled/Programmed/Protected-Unselected
Halted

in>FLASH arm

Arm FLASH for  loading

in>load my_appli.sx

Operation passed on FLASH_PAGE0 FLASH module

Halted

in>FLASH disarm

FLASH disarmed
```

7.5 Prepare and Program FLASH Memory

7.5.1 Non-Banked Memory Model

When programming in the normal memory model (non-banked memory range from 0x0000 to 0xFFFF), create a *.prm* file as usual and place code in the FLASH block area.

An example of programming in the assembler is:

```
LINK fibo_b32.abs
NAMES
       fibo.o
END

SEGMENTS
       MY_RAM = READ_WRITE 0x800 TO 0x87F;
       MY_FLASH = READ_ONLY 0x8000 TO 0xFBFF;
       MY_STK = READ_WRITE 0x880 TO 0x8FF;
END
PLACEMENT
       .data INTO MY_RAM;
       .text INTO MY_FLASH;
       .stack INTO MY_STK;
END
INIT main
VECTOR ADDRESS 0xFFFE main
```

An example when programming in C/C++ is:

```
LINK fibo_b32.abs

NAMES fibo.o start12s.o ansis.lib END
SECTIONS
       MY_RAM = READ_WRITE 0x800 TO 0x87F;
       MY_FLASH = READ_ONLY 0x8000 TO 0xFBFF;
       MY_STK = READ_WRITE 0x880 TO 0x8FF;
PLACEMENT
       DEFAULT_ROM INTO MY_FLASH;
       DEFAULT_RAM INTO MY_RAM;
       SSTACK INTO MY_STK;
END
VECTOR ADDRESS 0xFFFE _Startup
```

7.5.2 Banked Memory Model

When programming in the assembler, implement code in sections that will be mapped in the *.prm* file to the appropriate page. An example of source file code is:

```
XDEF Func1, Func2, main
Page1Code: section
Func1:
        ...
        RTS
Page2Code: section
Func2:
        ...
        RTS
UnpagedCode: section
main:
        ...
        CALL Func1,PAGE(Func1)
        CALL Func2,PAGE(Func2)
        ...
```

Assemble the file with the **Banked Memory Model** option selected in the MCUez assembler program. This option is located in the **Code Generation** tab of the **Option Settings** dialog box.

In the *.prm* file, sections (Page1Code, Page2Code) are placed in the PAGE_1 and PAGE_2 bank windows.

```
LINK my_appli.abs
NAMES
        my_appli.o
END

SECTIONS
        MY_RAM = READ_WRITE 0x2010 TO 0x23FF;
        MY_STK = READ_WRITE 0x2400 TO 0x24FF;
        NO_BANKED_ROM = READ_ONLY 0xC000 TO 0xFEFF;
        PAGE_1 = READ_ONLY 0x18000 TO 0x1BFFF;
        PAGE_2 = READ_ONLY 0x28000 TO 0x2BFFF;
PLACEMENT
        .data INTO MY_RAM;
        .text INTO NO_BANKED_ROM;
        .stack INTO MY_STK;
        Page1Code INTO PAGE_1;
        Page2Code INTO PAGE_2;
        UnpagedCode INTO NO_BANKED_ROM;
END
INIT main
VECTOR ADDRESS 0xFFFE main
```

For banked memory model, when programming in C/C++, link the application with the *ansib.lib* and *start12b.o* libraries.

The next example shows a *.prm* file for an HC12DG128 application, where the default ROM is in page 2 and page 4. Ensure that the code is properly located in a FLASH address range.

```
LINK my_appli.abs

NAMES my_appli.o ansib.lib start12b.o END
SECTIONS
       MY_RAM = READ_WRITE 0x2010 TO 0x23FF;
       MY_ROM = READ_ONLY 0xC000 TO 0xFEFF;
       PAGE_2 = READ_ONLY 0x28000 TO 0x2BFFF;
       PAGE_4 = READ_ONLY 0x48000 TO 0x4BFFF;
PLACEMENT
       _PRESTART, STARTUP,
       ROM_VAR, STRINGS,
       NON_BANKED, COPY INTO MY_ROM;
       DEFAULT_RAM INTO MY_RAM;
       MyPage, DEFAULT_ROM INTO PAGE_2, PAGE_4;
END
STACKSIZE 0x50
VECTOR ADDRESS 0xFFFE _Startup /* set reset vector IN FLASH on
_Startup */
```

To load, select ***FLASH...*** in the **SDI** target menu to open the **NVMC** dialog box.

If the user is sure about the application's absolute location, there is no need to select a block. However, if part or all of the program goes in a protected memory area (boot block), ensure that the matching block is unprotected (after reset, blocks are protected). For security reasons, unprotection is not done automatically and must be performed with the **Unprotect** button.

Click the **Load...** button (all blocks are automatically selected), then select the *.abs* file to load into FLASH. When loading has finished, the dialog box is refreshed with the new states of the blocks.

Close the dialog and run the application. On some hardware, it might be necessary to reset the target.

7.6 FLASH Memory Mapping

This section contains hardware-specific information about currently delivered MCU parameter files (*.fpp*).

7.6.1 M68EVB912B32 Evaluation Board Characteristics

- *.fpp* file name: *mcu03c1.fpp*
- FLASH blocks: 1
- FPP code loaded at 0x800, using 0x400 bytes
- Block name: FLASH_B32
- Block number: 0
- 32 Kbytes of FLASH located in 0x8000–0xFFFF or in 0x0000–0x7FFF (both handled according to MAPROM bit in MISC register)
- Boot sector unprotectable/protectable (2 Kbytes in range 0xF800–0xFFFF or in 0x7800–0x7FFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register)
- FLASH enable/disable via ROMON bit in MISC register

7.6.2 M68HC12A4 Evaluation Board Characteristics

- *.fpp* file name: *mcu03c3.fpp*
- FLASH blocks: 2
- FPP code loaded at 0x400, using 0x400 bytes

Block name: FEE28

- Block number: 0
- 28 Kbytes of FLASH located in 0x1000–0x7FFF or in 0x9000–0xFFFF (both handled according to MAPROM bit in MISC register)
- Boot sector unprotectable/protectable (8 Kbytes in range 0x6000–0x7FFF or 0xE000–0xFFFF) (via BOOTP bit in FEE28MCR register and LOCK bit in FEE28LCK register)
- FLASH enable/disable via ROMON28 bit in MISC register

Block name: FEE32

- Block number: 1
- 32 Kbytes of FLASH located in 0x8000–0xFFFF or 0x0000–0x7FFF (both handled according to MAPROM bit in MISC register)
- Boot sector unprotectable/protectable (8 Kbytes in range 0xE000–0xFFFF or 0x6000–0x7FFF) (via BOOTP bit in FEE32MCR register and LOCK bit in FEE32LCK register)
- FLASH enable/disable via ROMON32 bit in MISC register

7.6.3 HC12DG128/HC12DA128 Evaluation Board Characteristics

- *.fpp* filename: *mcu03c4.fpp*
- FLASH blocks: 10
- FPP code loaded at 0x2000, using 0x400 bytes
- All FLASH blocks enable/disable at the same time via ROMON bit in MISC register.

Block name: FLASH_4000

- Block number: 0
- 16 Kbytes unpaged FLASH located in 0x4000–0x7FFF and matches 32-K FLASH even page (6), (FLASH_PAGE6)

Block name: FLASH_PAGE0

- Block number: 1
- 16 Kbytes paged FLASH accessed in bank window 0x8000–0xBFFF, equivalent to FLASH 32-K even page (0)

Block name: FLASH_C000

- Block number: 2
- 16 Kbytes unpaged FLASH located in 0xC000–0xFFFF, and matches 32-K FLASH odd page (7), (FLASH_PAGE7)
- Boot sector unprotectable/protectable (8 Kbytes in range 0xE000–0xFFFF or paged range 0xA000–0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register)

Block name: FLASH_PAGE1

- Block number: 3
- 16 Kbytes paged FLASH accessed in bank window 0x8000–0xBFFF, equivalent to FLASH 32-K odd page (1)
- Boot sector unprotectable/protectable (8 Kbytes in range 0xA000–0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register)

Block name: FLASH_PAGE2

- Block number: 4
- 16 Kbytes paged FLASH accessed in bank window 0x8000–0xBFFF, equivalent to FLASH 32-K even page (2)

Block name: FLASH_PAGE3

- Block number: 5
- 16 Kbytes paged FLASH accessed in bank window 0x8000–0xBFFF, equivalent to FLASH 32-K odd page (3)
- Boot sector unprotectable/protectable (8 Kbytes in range 0xA000–0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register)

Block name: FLASH_PAGE4

- Block number: 6
- 16 Kbytes paged FLASH accessed in bank window 0x8000–0xBFFF, equivalent to FLASH 32-K even page (4)

Block name: FLASH_PAGE5

- Block number: 7
- 16 Kbytes paged FLASH accessed in bank window 0x8000–0xBFFF, equivalent to FLASH 32-K odd page (7)
- Boot sector unprotectable/protectable (8 Kbytes in range 0xA000–0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register)

Block name: FLASH_PAGE6

- Block number: 8
- 16 Kbytes paged FLASH accessed in bank window 0x8000–0xBFFF, equivalent to FLASH 32-K even page (6). Also equivalent to FLASH_4000 block

Block name: FLASH_PAGE7

- Block number: 9
- 16 Kbytes paged FLASH accessed in bank window 0x8000–0xBFFF, equivalent to FLASH 32-K odd page (7). Also equivalent to FLASH_C000 block.
- Boot sector unprotectable/protectable (8 Kbytes in range 0xA000–0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register)

7.7 FLASH Programming Examples

7.7.1 From a Command Line

In the following example, a program called *my_page5.sx* is loaded in the FLASH_PAGE5 block of an M68HC912DG128 CPU. Enter the `FLASH` command without parameters to display blocks and status.

```
in>FLASH
FLASH parameters loaded for M68HC912DG128 from
C:\MCUEZ\PROG\FPP\mcu03C4.fpp
MCU clock speed: 8025000

Module Addr RangeStatus
FLASH_40004000-7FFFEnabled/Blank - Unselected
FLASH_PAGE08000-BFFFEnabled/Blank - Unselected
FLASH_C000C000-FFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE118000-1BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE228000-2BFFFEnabled/Blank - Unselected
FLASH_PAGE338000-3BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE448000-4BFFFEnabled/Blank - Unselected
FLASH_PAGE558000-5BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE668000-6BFFFEnabled/Blank - Unselected
FLASH_PAGE778000-7BFFFEnabled/Blank/Protected - Unselected
HALTED
```

Unprotect block number 7 (FLASH_PAGE5) to load the application in this block.

```
in>FLASH unprotect 7
MCU clock speed: 8025000
Module Addr RangeStatus
FLASH_40004000-7FFFEnabled/Blank - Unselected
FLASH_PAGE08000-BFFFEnabled/Blank - Unselected
FLASH_C000C000-FFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE118000-1BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE228000-2BFFFEnabled/Blank - Unselected
FLASH_PAGE338000-3BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE448000-4BFFFEnabled/Blank - Unselected
FLASH_PAGE558000-5BFFFEnabled/Blank/Unprotected - Unselected
FLASH_PAGE668000-6BFFFEnabled/Blank - Unselected
FLASH_PAGE778000-7BFFFEnabled/Blank/Protected - Unselected
```

FLASH_PAGE5 is unprotected and ready to be selected and armed for programming.

```
in>FLASH select 7
in>FLASH arm
```

Next, load the application.

```
in>load a:\my_page5.sx
       RUNNING
```

Stop loading and disarm.

```
in>FLASH disarm
       FLASH disarmed
       Halted
```

Finally, display the state of the blocks with the `FLASH` command.

```
in>FLASH
MCU clock speed: 8025000
Module Addr RangeStatus
FLASH_40004000-7FFFEnabled/Blank - Unselected
FLASH_PAGE08000-BFFFEnabled/Blank - Unselected
FLASH_C000C000-FFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE118000-1BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE228000-2BFFFEnabled/Blank - Unselected
FLASH_PAGE338000-3BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE448000-4BFFFEnabled/Blank - Unselected
FLASH_PAGE558000-5BFFFEnabled/Programmed/Unprotected-Selected
FLASH_PAGE668000-6BFFFEnabled/Blank - Unselected
FLASH_PAGE778000-7BFFFEnabled/Blank/Protected - Unselected
HALTED
```

The FLASH_PAGE5 block is programmed. Protect the block and unselect it.

```
in>FLASH protect 7
MCU clock speed: 8025000
Module Addr RangeStatus
FLASH_40004000-7FFFEnabled/Blank - Unselected
FLASH_PAGE08000-BFFFEnabled/Blank - Unselected
FLASH_C000C000-FFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE118000-1BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE228000-2BFFFEnabled/Blank - Unselected
FLASH_PAGE338000-3BFFFEnabled/Blank/Protected - Unselected
FLASH_PAGE448000-4BFFFEnabled/Blank - Unselected
FLASH_PAGE558000-5BFFFEnabled/Programmed/Protected - Selected
FLASH_PAGE668000-6BFFFEnabled/Blank - Unselected
FLASH_PAGE778000-7BFFFEnabled/Blank/Protected - Unselected

in>FLASH unselect 7
```

7.7.2 From a Command File

This example automatically programs an application called *ffibo_32.abs* in FLASH on the HC12B32.

Create a command file (for example, *program.cmd*) that contains the following commands:

```
baud 57600
FLASH
FLASH select
FLASH unprotect
FLASH erase
FLASH arm
load ffibo_32.abs
FLASH disarm
FLASH protect
FLASH unselect
```

Call this file with the `call` command from the command line or in a debugger command file, such as *startup.cmd*.

An example is:

```
in>call program.cmd
executing program.cmd

!baud 57600
!FLASH
FLASH parameters loaded for M68HC912B32 from
```

```
c:\MCUEZ\PROG\FPP\mcu03C1.fpp

MCU clock speed: 8023000
Module Name Address Range Status
FLASH_B32   8000 - FFFF   Enabled/Programmed/Protected -
Unselected
HALTED
!FLASH select
!FLASH unprotect

MCU clock speed: 8023000
Module Name Address Range Status
FLASH_B32   8000 - FFFF   Enabled/Programmed/Unprotected -
Selected
!FLASH erase

MCU clock speed: 8023000
Module Name Address Range Status
FLASH_B32   8000 - FFFF   Enabled/Blank/Unprotected - Selected
!FLASH arm
Arm FLASH for loading.
!load ffibo_32.abs
executing ffibo_32.bpt

1!savebp off
RUNNING
done ffibo_32.bpt

!FLASH disarm
FLASH disarmed.
Halted
!FLASH protect

MCU clock speed: 8023000
Module Name Address Range Status
FLASH_B32    8000 - FFFF   Enabled/Programmed/Protected-Selected
HALTED
!FLASH unselect
!
done program.cmd

in>Help
```

Appendix A. Register Description File

A.1 Contents

A.2 Introduction

When loading a MCUez target, definitions of the I/O (input/output) registers are loaded from a *.reg* file. This allows the names of these registers to be used as parameters for commands or as operands in an expression. The syntax of the file is defined in **A.4 Description Using Extended Backus–Naur Form (EBNF)**.

There may be several different files depending on the MCU used. The name of the correct file is derived from the MCU identification number (MCU Id) in the following way:

MCUxxxx.REG

where *xxxx* is the MCU Id in hexadecimal representation. This file is expected to be found in the directory where the program files are located (for example, *..\PROG\REG*). If this file is not found, the *default.reg* file is searched for and loaded. If this file is not found, corresponding information will be missing and related commands may not deliver the complete results.

A.3 File Format

A header contains the name, identification number, and location of the register block of the MCU. The header is followed by a list of module descriptors. Each descriptor contains register definitions and (optionally) a memory map specification. The register definitions can be grouped under a group name. Each register definition defines the name, address and size of an I/O register. The memory map specification is used by the `MEM` command to display the configured memory of that module.

A.4 Description Using Extended Backus–Naur Form (EBNF)

The syntax of the register file is described here in EBNF.

```
MCUDescription              =  Header {Module}.
Header                      =  "MCU" McuName McuId RegBase
                               RegSize.
Module                      =  "MODULE" ModuleName {RegDef}
                               {GroupDef | MapDef}.
GroupDef                    =  "GROUP" GroupName {RegDef}.
RegDef                      =  RegName RegOffset Size.
MapDef                      =  "MEMMAP" BlkName BaseMapDef
                               {MapSecifier}.
BaseMapDef                  =  "BASE" Exp "SIZE" Exp
                               "ENABLED" Exp.
MapSpecifier                =  "SPECIFIER" [Label] Exp.
Exp                         =  CExpression | SwitchExpr.
SwitchExpr                  =  CExpression ":" {CaseSpec}.
CaseSpec                    =  "[" ConstValue ":"
                               (CExpression | StringDef) "]".
McuName                     =  StringDef.//name of the MCU
McuId                       =  ConstValue.//identification
                               number of the MCU
RegBase                     =  ConstValue.//base address of
                               the registers after reset
ModuleName                  =  Name.//name of the module
GroupName                   =  Name.//name of a group of
                               registers
RegName                     =  Name.//name of the register
RegOffset                   =  ConstValue.//offset from the
                               register base address
Size                        =  ConstValue.//size of the
                               register in bits
BlkName                     =  Name.//name of the memory
                               block
Label                       =  StringDef.//name to be used to
                               label the specifier
CExpression                 =  // expression defined in
                               ANSI-C that contains integers
ConstValue                  =  // constant value as defined
                               in ANSI-C
Name                        =   // identifier as defined
                               in ANSI-C
StringDef                   =  // any number of printable
                               characters in double quotes
```

[1] Evaluation of expressions are done with signed 32-bit arithmetic.
[2] Non-printable characters are interpreted as white spaces.

Example: This example describes a hypothetical MCU. It contains the modules ABC, SQIM, and FLASH. The SQIM module has two groups of registers, PORTS and CHIPSELECTS.

```
MCU "MY_MCU"  0x07A5  0xFFF000  0x1000
  MODULE ABC
    ABCMCR     0x700    16
    PORTABC    0x706    16
  MODULE SQIM
    SQIMCR     0xA00    16
    SYNCR      0xA04    16
    GROUP PORTS
      PORTA      0xA10     8
      PORTB      0xA11     8
    GROUP CHIPSELECTS
      CSPAR0     0xA44    16
      CSBARA     0xA60    16
      CSORA      0xA62    16
    MEMMAP CSA
      BASE (CSBARA & 0xFFF8) << 8
      SIZE CSBARA & 7 :
           [0:0x800] [1:0x2000] [2:0x4000]
          [3:0x10000] [4:0x20000] [5:0x40000]
           [6:0x80000] [7:0x80000]
         ENABLED (CSPAR0 & 3) >= 2
         SPECIFIER "ACCESS" (CSORA >> 11) & 3
:
                   [0:"None"][1:"Read"]
                   [2:"Write"][3:"Both"]
         SPECIFIER "BYTE" (CSORA >> 13) & 3 :
                   [0:"None"][1:"Lower"]
                   [2:"Upper"][3:"Both"]
         SPECIFIER (CSORA >> 4) & 3 :
                   [0:"None"][1:"Lower"]
                   [2:"Upper"][3:"Both"]
   MODULE FLASH
      FEEMCR     0x820    16
      FEEBAH     0x824    16
      FEEBAL     0x826    16
      MEMMAP FLASH
         BASE (FEEBAH << 16)
         SIZE  0x8000
         ENABLED (FEEMCR & 0x8000) == 0
<eof>
```

Appendix B. C Source-Level Debugging

B.1 Contents

B.2 Introduction

This appendix provides information on performing C source-level debugging (CSLD) with the MCUez debugger. The C source-level debugging capability is applicable only for applications that are compliant with the *ELF/DWARF 2.0* object format standard. The user's compiler must support this standard.

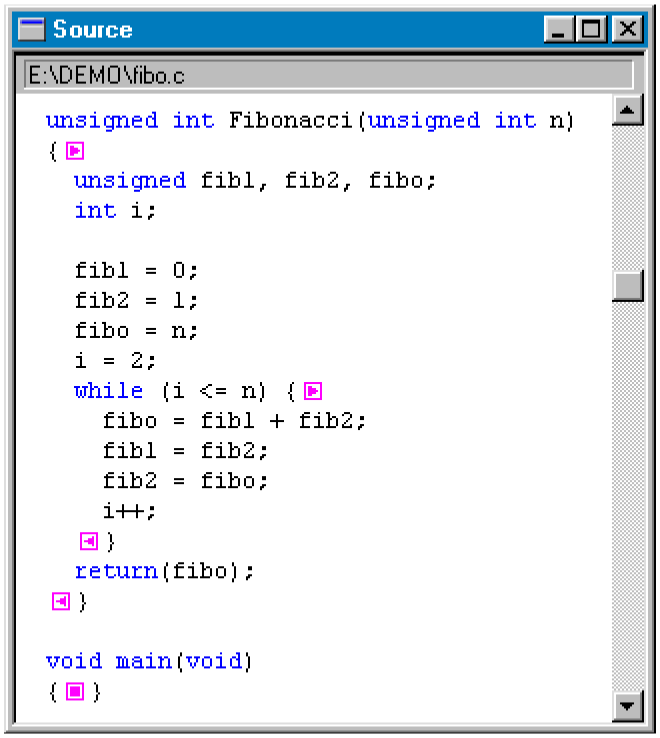
NOTE: *A license key is required to activate this feature. Contact HIWARE AG for information on CSLD pricing and how to obtain the license key.*

B.3 Source Component

The **Source** component window displays the source code of the program (application file). It enables the user to view, change, monitor, and control the current execution location in the program. The text displayed in the **Source** component window is chroma-coded. Language, keywords, comments, and strings are emphasized with different colors (respectively, blue, green, red). A word is selected by double clicking it.

Select a section of code by holding the left mouse button and dragging the mouse over the appropriate source code range. By clicking on the selection again and dragging it with the mouse to the assembly component, the corresponding assembly instructions are highlighted in the assembly component. Marks are displayed at all locations where breakpoints have been set. If execution has stopped, the current position is marked in the source component with the corresponding statement highlighted.

NOTE: *The text visible in the* ***Source*** *component window cannot be edited. The source component is a file viewer only.*


Source

E:\DEMO\fibo.c

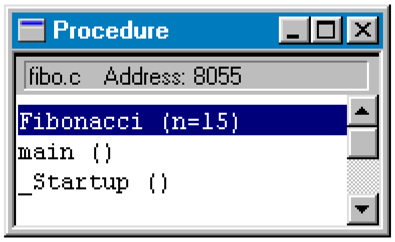
```
unsigned int Fibonacci(unsigned int n)
{
  unsigned fib1, fib2, fibo;
  int i;

  fib1 = 0;
  fib2 = 1;
  fibo = n;
  i = 2;
  while (i <= n) {
    fibo = fib1 + fib2;
    fib1 = fib2;
    fib2 = fibo;
    i++;
  }
  return(fibo);
}

void main(void)
{ }
```

Figure B-1. Source Component Window

B.4 Procedure Component

The procedure component displays the list of procedure or function calls that were performed up to the moment the program was halted. This is the procedure chain, also known as the call chain. Entries in the procedure chain are displayed in reverse order from the last call (most recent on top) to the first call (on bottom).

Procedure parameter values and types can be displayed. The object information bar contains the source module and the address of the selected procedure.


Procedure

fibo.c   Address: 8055

```
Fibonacci (n=15)
main ()
_Startup ()
```

Figure B-2. Procedure Component Window

B.4.1 Operations

Double clicking on a procedure name forces some open windows to display information about that procedure. The **Source** component window shows the procedure's source. The **Data** component window displays the local variables and parameters of the selected procedure. **Figure B-3** shows the **Procedure** component window menu.

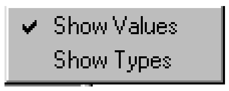


Figure B-3. Procedure Component Window Menu

- **Show Values** — Displays function parameter values in the procedure component
- **Show Types** — Displays function parameter types in the procedure component

B.4.2 Drag Out

The drag and drop actions in **Table B-1** are possible from the procedure component.

Table B-1. Procedure Component Drag and Drop Operations

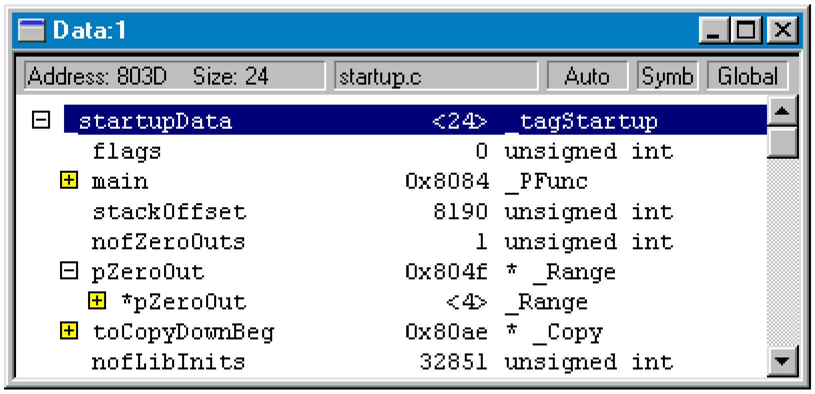
Destination Component	Action
Data | Local	Displays local variables from the selected procedure in the data component
Source	Displays source code of the selected procedure. Current instruction inside the procedure is highlighted in the source component.
Assembly	The current assembly statement inside the procedure is highlighted in the assembly component.

B.4.3 Drop Into

Nothing can be dropped into the **Procedure** component window.

B.5 Data Component

The **Data** component window contains the names, values, and types of global or local variables. The **Data** component window (in **Figure B-4**) shows all variables that are present in the current source module or procedure.



```
Address: 803D   Size: 24     startup.c
⊟ startupData            <24>  _tagStartup
    flags                   0  unsigned int
  ⊞ main               0x8084  _PFunc
    stackOffset          8190  unsigned int
    nofZeroOuts             1  unsigned int
  ⊟ pZeroOut           0x804f  * _Range
    ⊞ *pZeroOut           <4>  _Range
  ⊞ toCopyDownBeg      0x80ae  * _Copy
    nofLibInits         32851  unsigned int
```

Figure B-4. Data Component Window

The object info bar contains the address and size of the selected variable. It also contains the module name or procedure name where the displayed variables are defined, the display mode (automatic, locked, etc.), the display format (symbolic, hex, bin, etc.), and the current scope (global or local).

Various display formats such as symbolic representation, hexadecimal, octal, binary, signed, and unsigned can be selected. Structures can be unfolded to display their member fields. Pointers can be traversed to display the data they are pointing to. **Table B-2** lists the menu options for the data component.

Table B-2. Data Component Menu Options for C Source-Level Debugging

Menu Entry	Description
Zoom in	Develops the selected structure. The member field of the structure replaces the variable list.
Zoom out	Returns to previous level of development
Scope...	Switches between local or global variable display
Format...	Switches between Symbolic (display depends on type of variable), Hex (hexadecimal), Oct (octal), Bin (binary), Dec (signed decimal), UDec (unsigned decimal) display format
Mode...	Switches between automatic, periodical, locked, or frozen update mode

In automatic mode (default), variables are updated when the target is stopped. Variables from the currently executed module or procedure are displayed in the data component.
In locked and frozen mode, variables from a specific module are displayed in the data component. In that case, the same variables are always displayed in the data component.
In locked mode, variable values displayed in the data component are updated when the target is stopped.

In frozen mode, values displayed in the data component are not updated when the target is stopped.

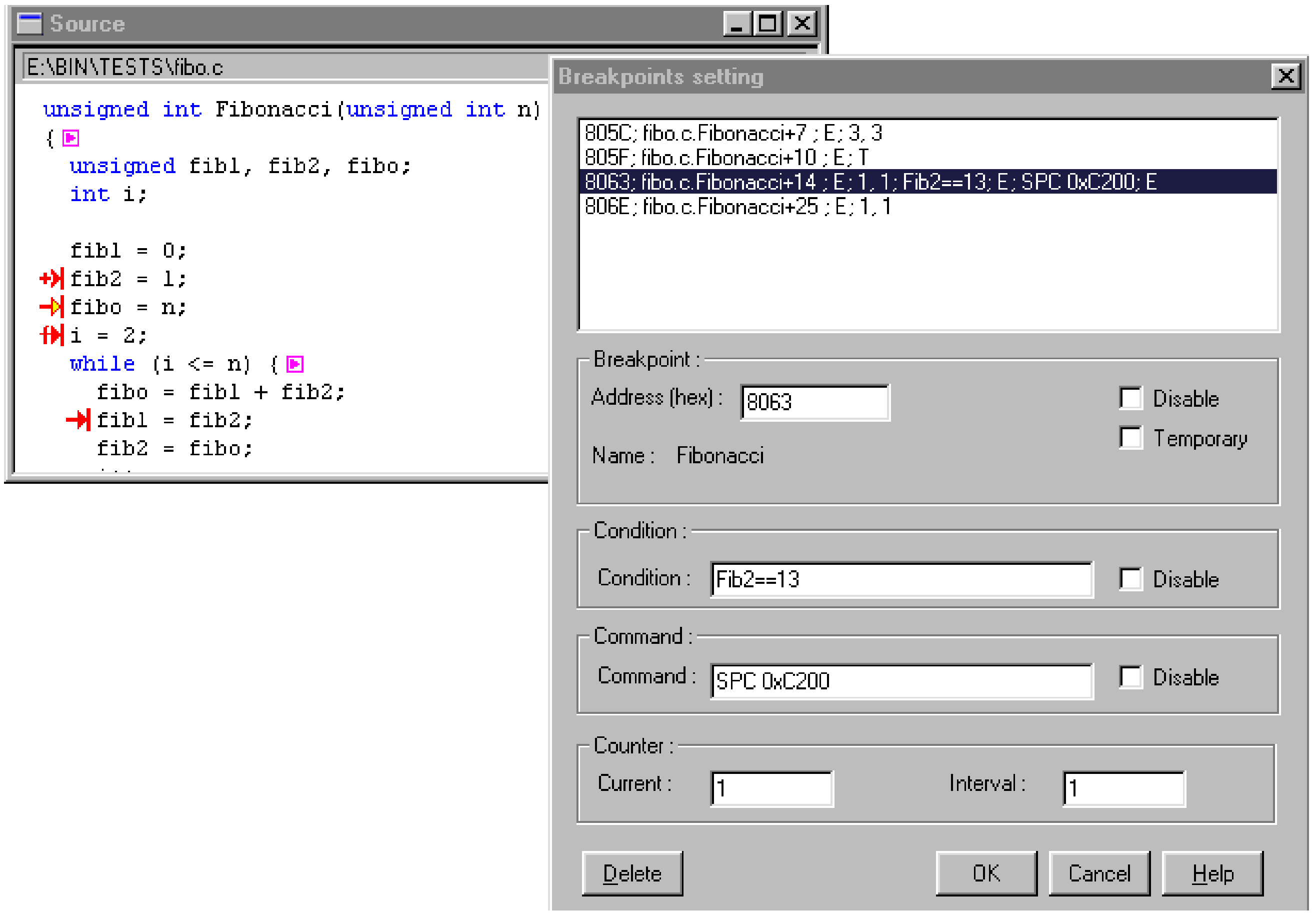
In periodical mode, variables are updated at regular time intervals when the target is running. The default update rate is 1 second.

B.6 Breakpoints Setting Dialog

The **Breakpoints** setting dialog box consists of:

- A list box that displays currently defined breakpoints
- A **Breakpoint:** group box that displays the address of the currently selected breakpoint, name of the procedure in which the breakpoint has been set, state of the breakpoint (disabled or not), and type of breakpoint (temporary or permanent)

- A **Counter:** group box that displays the current value and interval value of the counter. This group allows the user to define a counting breakpoint (See **B.10 Stopping an Application**).
- A **Delete** button to remove the currently selected breakpoint
- **OK** button to validate all modifications
- **Cancel** button to disregard all modifications and retain previous values
- **Help** button to open the help file



E:\BIN\TESTS\fibo.c

```
unsigned int Fibonacci(unsigned int n)
{
  unsigned fib1, fib2, fibo;
  int i;

  fib1 = 0;
  fib2 = 1;
  fibo = n;
  i = 2;
  while (i <= n) {
    fibo = fib1 + fib2;
    fib1 = fib2;
    fib2 = fibo;
    ...
```

```
805C; fibo.c.Fibonacci+7 ; E; 3, 3
805F; fibo.c.Fibonacci+10 ; E; T
8063; fibo.c.Fibonacci+14 ; E; 1, 1; Fib2==13; E; SPC 0xC200; E
806E; fibo.c.Fibonacci+25 ; E; 1, 1
```

Breakpoint :
Address (hex) : 8063 — Disable [ ] — Temporary [ ]
Name : Fibonacci

Condition :
Condition : Fib2==13 — Disable [ ]

Command :
Command : SPC 0xC200 — Disable [ ]

Counter :
Current : 1 — Interval : 1

Figure B-5. Breakpoints Setting Dialog Window

B.7 General Rule for Halting on a Control Point

Counting Control Point: If the interval is greater than one (1), a counting control point has been defined. When the target application is running, its current value is decremented each time the control point is reached. The debugger will halt on this control point if the control point counter is equal to zero (0). The interval must be greater than zero.

B.8 Configuring the Default Layout

This line must be in the *project.ini* file to establish a default layout for the MCUez debugger:

```
Window9=Procedure   0  30  50  15
```

B.9 Loading an Application

The global data component contains the list of global variables defined in the module that contains the application entry point. The local data component might be empty.

B.10 Stopping an Application

The **Data** component window (assigned the global attribute) displays the name and value of all global variables defined in the module where the currently executed procedure is implemented. The module name is specified in the data component information bar. The **Data** component window (assigned the local attribute) displays the name and value of the local variables defined in the current procedure. The name of the procedure is specified in the data information bar.

B.11 Defining Counting Breakpoints

Counting breakpoints are activated after the instruction has been executed a specified number of times. This section describes breakpoint operations.

A counting breakpoint is recognized by this icon: ➔| .

A counting breakpoint is set by the **Breakpoints Setting** dialog. To access this dialog box:

- Point to a C statement in the source component window, hold the left mouse button, and press the S key.
- Point at a C statement in the **Source** component window and click the right mouse button to open the **Source** pop-up menu, then select **Set BreakPoint** or **Show BreakPoints**.
- Select **Run | Breakpoints ....** from the main menu bar.

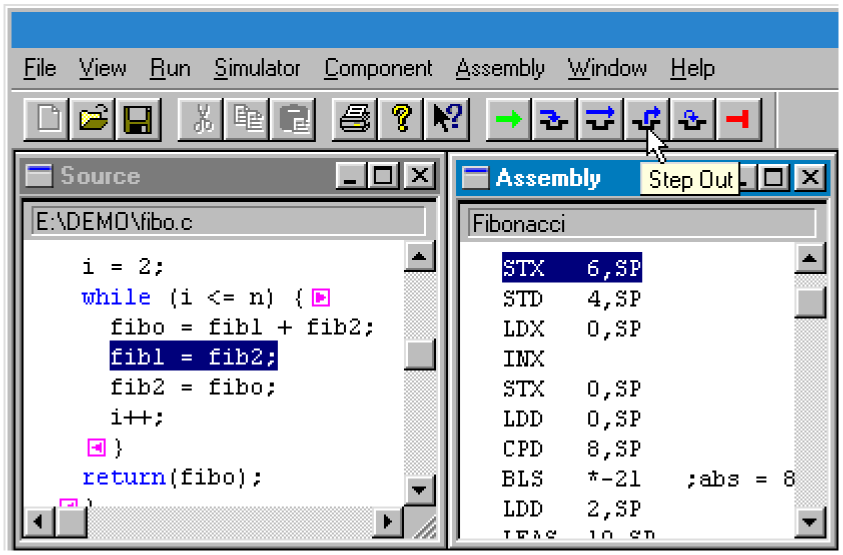
If program execution continues, the **Current** field is decremented each time the instruction containing the counting breakpoint is reached. When **Current** is equal to 0, the application stops. If the check box **Temporary** is not checked (not a temporary breakpoint), **Current** is reloaded with the value stored in **Interval** to enable the counting breakpoint again.

B.12 Stepping in the Application

The MCUez debugger provides stepping functions at the application source level and assembler level.

B.12.1 Stepping at Source Level

Figure B-6 shows a typical stepping-at-source-level operation.



```
  i = 2;
  while (i <= n) {
    fibo = fib1 + fib2;
    fib1 = fib2;
    fib2 = fibo;
    i++;
  }
  return(fibo);
```

```
STX   6,SP
STD   4,SP
LDX   0,SP
INX
STX   0,SP
LDD   0,SP
CPD   8,SP
BLS   *-21    ;abs = 8
LDD   2,SP
```

Figure B-6. Stepping-at-Source Level Window

The debugger provides two ways of stepping to the next source instruction:

1. Select **Run | Single Step**.
2. Click the **Single Step** icon on the debugger toolbar .

STEPPED displayed in the status line indicates that the application is stopped by a step function.

If the application was previously stopped on a function invocation, a **Single Step** stops the application at the beginning of the invoked function.

The display in the assembly component is always synchronized with the display in the source component. The highlighted instruction in the assembly component is the first assembler instruction generated by the highlighted instruction in the source component.

Elements from the register, memory, or data components that are displayed in red are the register, memory position, and local or global variables. The indicated variables are those whose values have changed during execution of the source statement.

B.12.2 Stepping Over a Function Call (Flat Step)

The debugger provides two ways of stepping over a function call:

1. Select **Run | Step Over**.
2. Click the **Step Over** icon on the toolbar .

`STEPPED OVER` displayed in the status line indicates that the application was stopped by a step over function.

If the application was previously stopped on a function invocation, a **Step Over** halts the application on the source instruction directly following the function invocation.

The display in the assembly component is always synchronized with the display in the source component. The highlighted instruction in the assembly component is the first assembler instruction generated by the highlighted instruction in the source component.

Elements from the register, memory, or data components that are displayed in red are the register, memory position, and local or global variables. The indicated variables are those whose values have changed during execution of the source statement.

B.12.3 Stepping Out of a Function Call

The debugger provides two ways of stepping out of a function call:

1. Select **Run | Step Out**
2. Click the **Step Out** icon on the toolbar .

`STOPPED` displayed in the status line indicates that the application is stopped by a step out function.

If the application was previously stopped on a function invocation, a **Step Over** halts the application on the source instruction directly following the function invocation.

B.13 Displaying a Local Variable from a Function

The debugger provides two methods to view the local variable list defined in a function.

1. Using drag and drop — Drag a function name from the procedure component to a data component with attribute local.
2. Using double click — Double click a function name in the procedure component.

The data component (with attribute local that is neither frozen nor locked) displays the list of variables (with their values and type) defined in the selected function.

B.14 Miscellaneous C Source-Level Commands

This section describes all debugger commands associated with C source-level debugging.

SPROC

Short description:

Shows information associated with the specified procedure

Syntax:

```
SPROC level
```

Description:

In the data component, the `SPROC` command shows local variables of the corresponding procedure stack level.

In the source component, the `SPROC` command shows the corresponding module’s source text, scrolls to the corresponding procedure, and highlights the statement that is in the procedure chain.

`level = 0` is the current procedure level. `level = 1` is the caller stack level and so on.

Data component example:

```
in>Data:2 < SPROC 0
```

This command displays the local variables defined in the caller function number 1 in the call chain.

Source component example:

```
in>Source < SPROC 1
```

ATTRIBUTES

Short description:

Sets the display inside a component window

In the Procedure Component

Syntax:

`ATTRIBUTES list`

Syntax:

where `list=command{,command}`

command=`VALUES (ON|OFF)`| `TYPES (ON|OFF)`

Description:

The `ATTRIBUTES` command sets the display and state options of the **Procedure** component window.

The `VALUES` and `TYPES` command `ON` or `OFF` indicates if the values or types should be displayed in the **Procedure** window. This command is applicable for the procedure component only when performing C source-level debugging.

Example:

Procedure < ATTRIBUTES VALUES ON,TYPES ON

Parameter types and values are displayed in the **Procedure** component window.

In the Data Component

Syntax:

`ATTRIBUTES list`

Arguments:

`list=command{,command})`

command=`FORMAT(bin |oct |hex |signed |unsigned |symb) |MODE(automatic |periodical |locked |frozen)` | `SCOPE` (global | local) | `SPROC` module | `SMOD` module | `UPDATERATE` rate

Description:

The `ATTRIBUTES` command sets the display and state options of the **Data** component window.

The `FORMAT` command indicates how variables will be represented. Display formats are binary, octal, hexadecimal, signed decimal, unsigned decimal, or symbolic.

The `SCOPE` command selects and displays global or local variables.

The `MODE` command selects the display mode of variables.

In automatic mode (default mode), variables are updated when the target is stopped. Variables from the currently executed module or procedure are displayed in the data component.

In locked and frozen mode, variables from a specific module are displayed in the data component.

In locked mode, values from variables are updated when the target is stopped.

In frozen mode, values are not updated when the target is stopped.

In periodical mode, variables are updated at regular time intervals when the target is running. The default update rate is 1 second.

The `UPDATERATE` command sets the update rate for the data component. This command is only relevant when the update mode for the data component is set to periodical.

The `SMOD` (show module) command displays global variables of the corresponding module.

The `SPROC` (show procedure) command displays local variables of the procedure.

Arguments:

`Hex`	Sets format representation to hexadecimal
`Oct`	Sets format to octal
`Bin`	Sets format to binary
`Symb`	Sets format as a symbol
`Signed`	Displays value in signed decimal format
`Unsigned`	Displays value in unsigned decimal format
`Periodical`	Set data component to periodical update mode.

`Locked`	Set data component to locked update mode.
`Frozen`	Set data component to frozen update mode.
`Automatic`	Set data component to automatic update mode.
`Module`	Specified module
`Rate`	Update rate in tenths of a second. Valid values for the rate are 0 to 600.

Equivalent operations:

`ATTRIBUTES FORMAT`	Select menu entry **Data | Format...**
`ATTRIBUTES MODE`	Select menu entry **Data | Mode...**
`ATTRIBUTES SCOPE`	Select menu entry **Data | Scope...**
`ATTRIBUTES SPROC`	Drag and drop from procedure component to data component.
`ATTRIBUTES SMOD`	Drag and drop from module component to data component.
`ATTRIBUTES UPDATERATE`	Select menu entry **Data | Mode | Periodical**.

Example:

```
Data < ATTRIBUTES MODE FROZEN
```

In the data component, the mode for updating global variables is set to frozen. Variables are not refreshed when the application is halted.

Appendix C. Extended Backus-Naur Form (EBNF)

C.1 Contents

C.2 EBNF File Format

This section gives a short introduction to EBNF and an overview of EBNF notation.

Extended Backus–Naur Form (EBNF) is frequently used in this user’s manual to describe file formats and syntax rules.

C.3 EBNF Example

This is an example of EBNF notation:

```
ProcDecl=PROCEDURE "(" ArgList ")".
ArgList=Expression {"," Expression}.
Expression=Term ("*"|"/") Term.
Term=Factor AddOp Factor.
AddOp="+"|"-".
Factor=(["-"] Number)|"(" Expression ")".
```

The EBNF language can be used to express the syntax of context-free languages. EBNF is a set of rules called *productions* of the form:

```
LeftHandSide=RightHandSide
```

The left hand side is a non-terminal symbol, and the right hand side describes how it is composed.

EBNF consists of the following symbols:

- Terminal symbols (terminals for short) are the basic symbols which form the language described. In the previous example, the word `PROCEDURE` is a terminal. Punctuation symbols of the language described (not EBNF) are quoted (they are terminals, too), while other terminal symbols are printed in **boldface**.
- Non-terminal symbols (non-terminals) are syntactic variables and have to be defined in a production, for example, they have to appear on the left hand side of a production. In the previous example, there are many non-terminals, for instance, `ArgList` or `AddOp`.
- The vertical bar (`|`) denotes an alternative, for example, either the left or right side of the bar must appear in the language described. For example, the third production above means "an expression is a term followed by either an asterisk (`*`) or a slash (`/`) followed by another term."
- Parts of an EBNF production enclosed by square brackets (`[` and `]`) are optional. They may appear one time in the language, or they may be skipped. The minus sign in the last production is optional, both `–7` and `7` are allowed.
- The repetition is another useful construct. Any part of a production enclosed by curly brackets (`{` and `}`) may appear any number of times in the language described (including 0, for example, it may also be skipped). `ArgList` is an example. An argument list is a single expression or list of expressions separated by commas. (Note that the syntax in the example does not allow empty argument lists.)
- For readability, normal parentheses may be used for grouping EBNF expressions, as done in the last production of the example. Note the difference between the first and second left bracket. The first one is part of EBNF itself, the second one is a terminal symbol (it is quoted) and, therefore, may appear in the language described.
- A production is always terminated by a period (.).

C.4 EBNF Syntax

The following defines EBNF syntax:

```
Production=NonTerminal "=" Expression ".".
Expression=Term {"|" Term}.
Term=Factor {Factor}.
Factor=NonTerminal
| Terminal
| "(" Expression ")"
| "[" Expression "]"
| "{" Expression "}".
Terminal=Identifier | “"“ <any char> “"“.
NonTerminal=Identifier.
```

The identifier for a non-terminal can be any name. Terminal symbols are either identifiers appearing in the language described or any character sequence that is quoted.

C.5 Extensions

In addition to this standard definition of EBNF, the following notational conventions are used:

- Count repetition — Anything enclosed by curly brackets ({ and }) and followed by a superscripted expression *x* must appear exactly *x* times. *x* may also be a non-terminal. In this example, exactly four stars are allowed:

  `Stars={"*"}`^{4}

- The size in bytes — Any identifier immediately followed by a number *n* in square brackets ([and]) may be assumed to be a binary number with the most significant byte stored first, having exactly *n* bytes. Example:

  `Struct=RefNo FilePos[4]`

- In some examples, text is enclosed by greater than and less than symbols (< and >). This text is a meta–literal. Whatever the text says may be inserted in place of the text. <any char> in the example in **C.4 EBNF Syntax** means any character can be inserted.

Appendix D. Serial Device Interface (SDI)

D.1 Contents

D.2 Introduction

The serial device interface (SDI) is an interface developed by Motorola and used by MCUez to communicate with an external target system.

D.3 Overview

With this interface, an executable program is downloaded from the MCUez environment to an external target system based on a Motorola MCU, which will execute the program. Feedback from the target system to MCUez is provided.

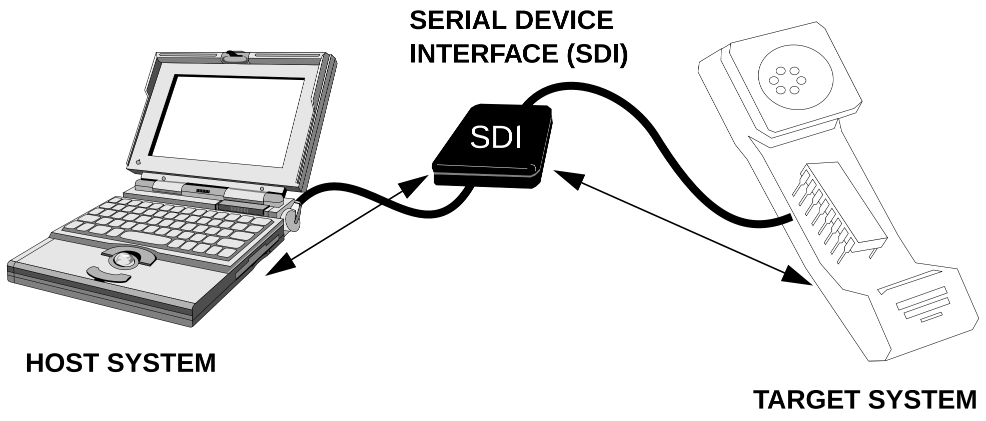


Figure D-1. Serial Device Interface (SDI)

MCUez supervises and monitors the target system's MCU. The user can read and write to internal/external memory (even when the CPU is running), single-step, run and stop the CPU, and set breakpoints in the code.

The SDI interface is designed around a serial communication link. The interface is supported by any communication device on the system. The SDI target driver handles the communication protocol between the SDI and the system. However, the user can adapt the target system to the SDI interface. The SDI serial port

6-pin connector for the HC12 target system is shown in **Figure D-2**. The SDI interface includes two 6-pin serial port connectors.

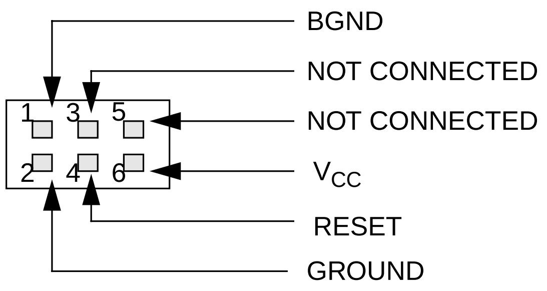


Figure D-2. Serial Port Connector for the HC12

NOTE: *The serial port connector for the CPU16/32 is shown in* ***Figure D-3****. The HC16 and 683xxx chip series use a 10-pin connector.*

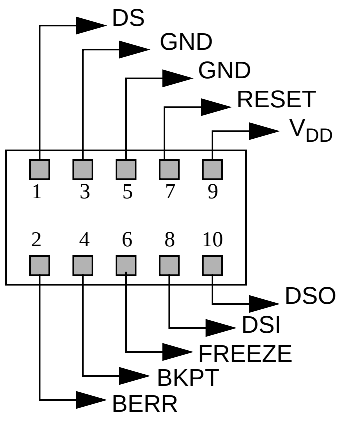


Figure D-3. Serial Port Connector for the CPU16/32

The communication protocol is defined in Section 8 Development and Debug Support of the *CPU12 Reference Manual*, Motorola document order number CPU12RM/AD.

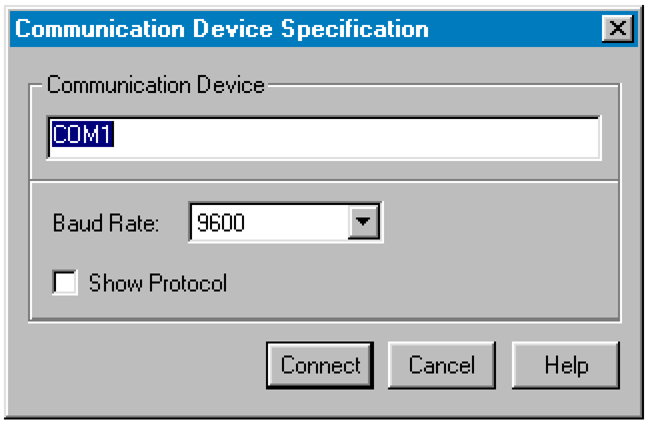
D.3.1 System Power

The target system supplies power to the SDI. This power supply must conform to the TTL (transistor transistor logic) standard. If it does not conform to the standard, the SDI must have its own power supply.

D.4 Communication Configuration

Communication between the MCUez debugger and SDI interface is set automatically.

If the host and target are not connected, a dialog box pops up in the debugger as shown in **Figure D-4**.


Communication Device Specification

Communication Device: COM1

Baud Rate: 9600

☐ Show Protocol

Figure D-4. Communication Device Dialog Box

Enter an available port in the **Communication Device** edit box. Select the baud rate and click **Connect**.

If the connection is not established, an error message is displayed. Click **OK** to close the error message and try another port. Once a connection is made, the settings are saved. The default device is COM1.

NOTE: *The communication device and baud rate saved by this dialog box override the* `BAUDRATE` *and* `COMDEV` *environment variables in the default.env file.*

D.4.1 Data Format

The data format used is eight data bits, one stop bit, no parity, and variable baud rate. The default speed is 9600 baud unless redefined by selecting the **SDI | Communication...** menu option in the **Target** menu.

D.5 Default Target Setup

The **Target** menu loads the SDI target component. It can also be set as the default target in the *project.ini* file. The *project.ini* file is located in the working directory. **Figure D-5** shows an example of a *project.ini* file.

```
[DEFAULTS]
 Window0=Source     0   0  50  40
 Window1=Assembly  50   0  50  40
 Window2=Register  50  40  50  30
 Window3=Memory    50  70  50  30
 Window4=Data       0  40  50  25
 Window5=Command    0  65  50  20
 Window6=Module     0  85  50  15
 Target=MotoESL
```

Figure D-5. Example of *project.ini* File

D.6 Setting the Target

Load the SDI target component by opening the **Component** menu and selecting **Set Target**. Then select the MotoESL target component.

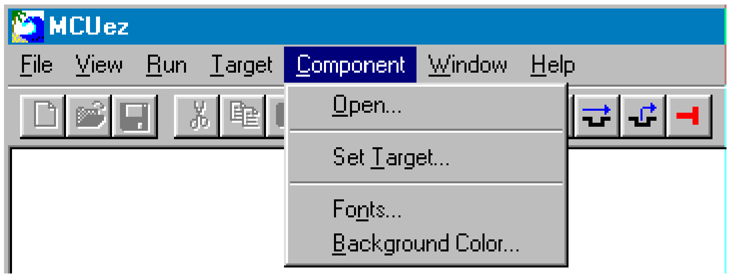


Figure D-6. Selecting MotoESL Target Component

After loading the MotoESL target component, the **Target** menu item is replaced by the **SDI** menu.

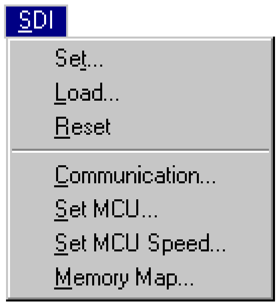


Figure D-7. SDI Target Menu

D.6.1 SDI Status Bar

After the SDI target component has been loaded, the debugger status bar displays the current status of the SDI target. Reading from left to right is the baud rate, debugger running mode, clock frequency of target, power mode, current MCU Id, and SDI operation.

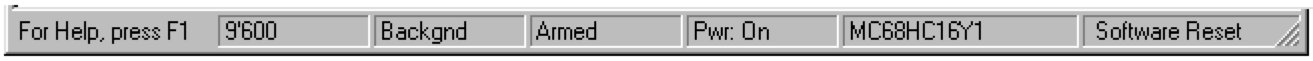


Figure D-8. SDI Status Bar

D.7 Advanced SDI Environment Setup

Communication between MCUez and SDI is automatically set at startup. However, it is possible to manually set communication and other parameters as described in the following sections.

D.7.1 SDI Default Environment

Parameters for the SDI target component are set in the *default.env* file located in the working directory. The user can change the default values as needed.

D.7.2 IMODULE

The MCU Id provides the default values for the integration module. To override these values, specify one of the following module types with the `IMODULE` parameter.

```
SIM, SCIM, RPSCIM, SCIM2, LIM_N_MUX, LIM_MUX,
SLIM_N_MUX, SLIM_MUX
```

Example:

```
IMODULE=SIM
```

D.7.3 COMDEV

This parameter specifies the communication device to use between the host and SDI. `COM1` is the default communication device for PCs. If necessary, another valid device can be set to establish communications.

Example:

```
COMDEV=COM3
```

D.7.4 SDI Target Startup File

The *startup.cmd* file is executed by the MCUez debugger after the SDI target driver is loaded. This file must be located in the working directory. The user can specify any MCUez debugger command in this file.

Example of a *startup.cmd* file:

```
wb 0x0035  0x00
wb 0x0012  0x11
```

D.7.5 SDI Reset Command File

The *reset.cmd* file is executed when the debugger is launched or when **Reset** is selected in the SDI menu. This file must be located in the working directory. The user can specify any MCUez debugger command in this file.

D.8 SDI Target Menu

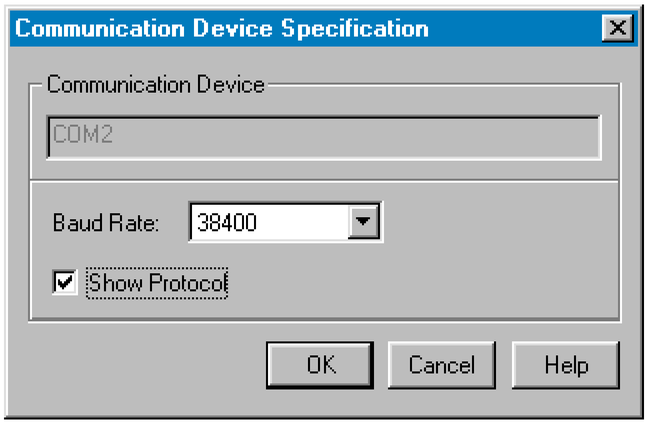
D.8.1 Loading an Application

Select the **SDI | Load...** menu option to load the application to be debugged (for example, an *.abs* file).

D.8.2 Communications Baud Rate

The baud rate is set automatically when the debugger starts to communicate with the SDI. However, the user can modify this baud rate.

Select the **SDI | Communication...** menu option to display the **Communication** dialog box shown in **Figure D-9**. If the maximum rate the host computer supports is known, select it from the drop down menu (115,200 is not supported with SDI); otherwise, select 57600. If communication fails, the baud rate reduces automatically until communication is established.



COM2

Baud Rate: 38400

☑ Show Protocol

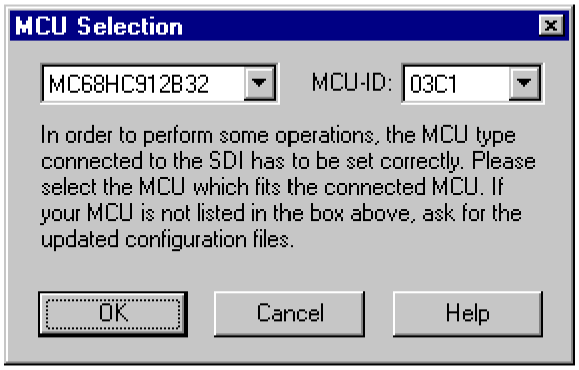
Figure D-9. Communication Device Dialog Box

The maximum baud rate depends on the speed and interrupt load of the host computer. On slow notebook computers or on computers running on a network, the maximum baud rate can be as low as 19,200. A buffered I/O card allows the maximum rate of 57,600 on any host computer. The default value is 9600.

If the **Show Protocol** box is checked, all commands and responses sent and received are reported in the **Command Line** window. This feature is used by support personnel from Motorola.

D.8.3 MCU Selection

Select the **SDI | Set MCU...** menu option to open this dialog box.



MC68HC912B32 MCU-ID: 03C1

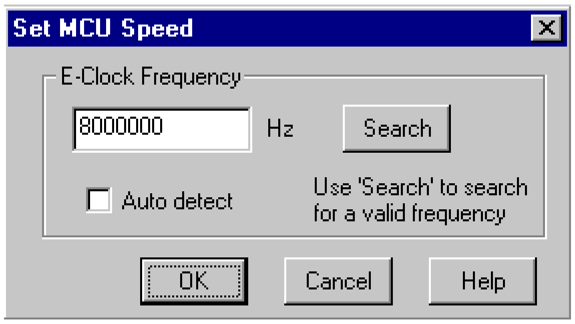
In order to perform some operations, the MCU type connected to the SDI has to be set correctly. Please select the MCU which fits the connected MCU. If your MCU is not listed in the box above, ask for the updated configuration files.

Figure D-10. MCU Selection Dialog Box

In this dialog box, select the MCU currently used. There are two drop down menus. They show the currently selected MCU name and MCU Id. Information is taken from the *mdsemcu.ini* file. If a specific MCU is not found in this file, the user must obtain the latest version of this file.

D.8.4 MCU E-Clock Frequency

Select the **SDI | Set MCU Speed...** menu option to open this dialog.



8000000 Hz

☐ Auto detect

Use 'Search' to search for a valid frequency

Figure D-11. Setting the MCU Speed

This dialog shows the current setting of the E-clock frequency to be used by the MCU. This frequency must be known by the SDI for proper communication through the BDM (background debug mode). This is typically half of the crystal oscillator frequency for the CPU12.

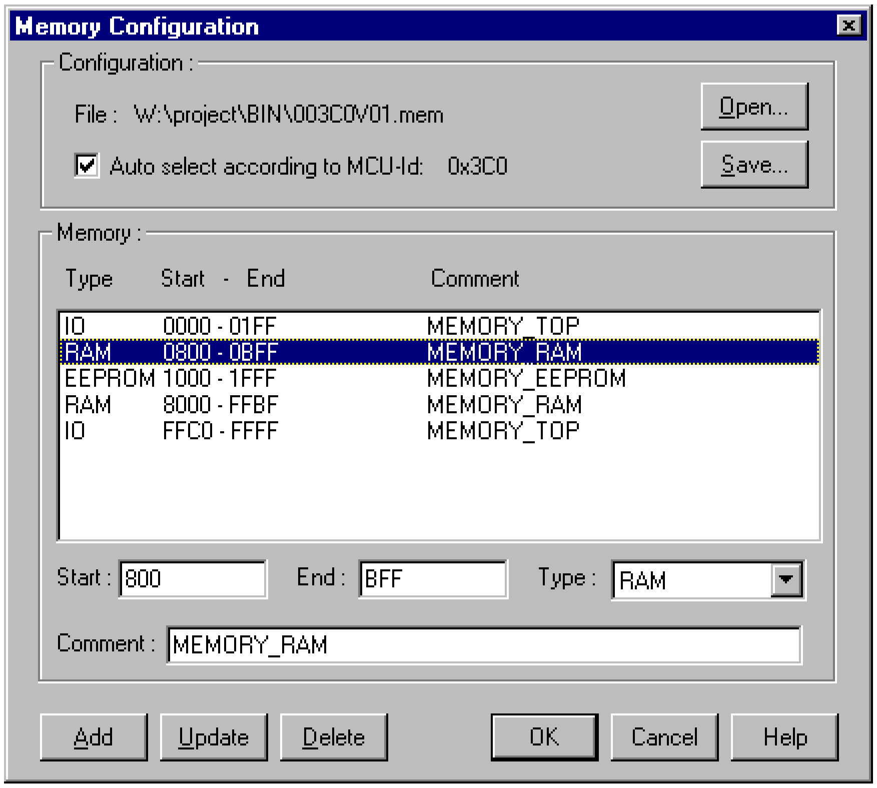
In the edit box, specify the frequency to be used. Click **Search** to verify communication. If it fails, a valid frequency is searched for in the following order:

> 16 MHz, 8 MHz, 4 MHz, 2 MHz, 1 MHz, 12 MHz, 6 MHz, 3 MHz, 1.5 MHz, 14 MHz, 10 MHz, 7 MHz, 5 MHz

At startup, the debugger uses the specified frequency. If the debugger fails and **Auto detect** is checked, a valid frequency is searched. If communication is not established, an error message is displayed.

D.8.5 Memory Configuration

Select the **SDI | Memory Map...** menu option to open this dialog.



File : W:\project\BIN\003C0V01.mem

[✓] Auto select according to MCU-Id: 0x3C0

Type	Start - End	Comment
IO	0000 - 01FF	MEMORY_TOP
RAM	0800 - 0BFF	MEMORY_RAM
EEPROM	1000 - 1FFF	MEMORY_EEPROM
RAM	8000 - FFBF	MEMORY_RAM
IO	FFC0 - FFFF	MEMORY_TOP

Start : 800 End : BFF Type : RAM

Comment : MEMORY_RAM

Figure D-12. Memory Configuration Dialog Box

The memory configuration dialog contains the physical setup of the target. This setup loads automatically if the **Auto select** box is checked. MCUez identifies the setup through the MCU Id given in the previous dialog. However, the user can modify this configuration, save it, or load a different one. The user can also add new fields, edit, or remove fields.

D.9 SDI Operations

D.9.1 On-Chip Hardware Breakpoint

An on-chip hardware breakpoint module can be used to implement breakpoints. To invoke this module, initialize the environment variable `HWBPMODULEADR` in the *default.env* file with the address of the hardware breakpoint module. If hardware breakpoints and the associated module are not being used, comment out the line shown in this example.

Example:

```
HWBPMODULEADR=0x20
```

The SDI hardware can only handle two breakpoints at the same time. Additional breakpoints will be considered as software breakpoints. When debugging code in FLASH, do not set more than two breakpoints.

Some actions like “stepping over” or “stepping out” use one internal breakpoint. For efficient operation, reduce the number of hardware breakpoints to one.

D.9.2 EEPROM Programming

To download code or data into the EEPROM, MCUez needs the location of this EEPROM. To identify the location, define these two environment variables.

`EEPROM_START`	Defines the address of the first byte of the EEPROM memory
`EEPROM_END`	Defines the address of the last byte of the EEPROM memory

Example:

```
EEPROM_START=0x0D00
EEPROM_END=0xFFF
```

This specifies the memory range of the EEPROM as 0xD000 to 0x0FFF. When writing to these addresses, the EEPROM is automatically programmed to download a program or modify the memory or variables interactively.

D.10 OperatingEVBwithSDI

The SDI can be used with any target system equipped with background debug mode (BDM). The SDI also operates with the M68HC812A4EVB and M68HC912B32EVB HC12 evaluation boards (EVB). The evaluation boards support the HC124A and HC129B32 controllers.

D.10.1 Operating the SDI with the MC68HC812A4EVB

To operate the SDI with the M68HC12A4EVB, remove the BGND jumper in the EVB and disable the ROM chip (monitor program) by removing the chip select jumper from the ROM socket. Also ensure that both operating modes (MODA and MODB) are set to low.

However, the user can replace the default ROM chips with RAM chips. Consult the *M68HC12A4EVB User's Manual*, Motorola document order number HC12A4EVBUM/D, to get the correct setup for the chip select jumper and other jumpers on the EVB.

NOTE: *When connecting the SDI cable to the EVB, ensure that the red-colored side of the cable is aligned with the odd-numbered connector pins on the board.*

D.10.2 Operating the SDI with the M68HC912B32EVB

To run the SDI with the M68HC912B32EVB, connect the SDI cable to the BDM IN connector on the evaluation board. Set the operating mode as EVB or PAD. When selecting the PAD operating mode, the SDI can be used in debugging procedures on the target system. For detailed information, see the *M68HC912B32EVB User's Manual*, Motorola document order number 68EVB912B32UM/D.

D.10.3 Demo Programs

To run the MCUez demo with the M68HC12A4EVB, place the jumper in the CSPO pins 2 and 3 on the RAM socket to allocate memory from $8000. All demo programs delivered on the SDI installation disk can be loaded and run with SDI on the EVB board.

D.11 Periodic Updates

The debugger can be configured to periodically refresh the data and memory components while the application is running. Since the emulator ensures non-intrusive access on emulated RAM, the application continues to run in real time.

For the debugger to periodically update data in the memory or data component window:

- Click the right mouse button in the memory or data component window and select **Mode | Periodical** to open the **Update Rate** dialog box.
- Enter 10 in the rate edit box and click **OK**. Information will refresh every second.
- Start the application. The memory or data component is periodically refreshed during program execution.

NOTE: *Due to hardware restrictions, periodic updates are not possible when hardware breakpoints or triggers are used in the emulator.*

D.12 SDI Commands

SDI incorporates two commands: `BAUD` and `RESET`.

BAUD	
Short:	baud rate
Syntax:	`BAUD [rate]`
	`rate:` Specifies the new baud rate. It must be one of the following values: 1200, 2400, 4800, 9600, 19200, 28800, 38400, 57600

Description:

The `BAUD` command sets or displays the communication baud rate between the system controller and host computer. For maximum performance, set the baud rate as high as the host computer can accommodate. The maximum rate is 57,600; the default baud rate is 9600.

Enter the `BAUD` command without specifying a value to display the **Communications Specification** dialog box. If the host is unable to support the requested baud rate, an "Out of synchronization" message is displayed. Select **ABORT** to exit or **RETRY** to use the default baud rate.

Example:

```
BAUD 57600
```

RESET	
Short:	target reset
Syntax:	`RESET`

Description:

Resets the SDI target and executes the *reset.cmd* file

Index

Need to know more? That’s ez, too.

Technical support for MCUez development tools is available through your regional Motorola office or by contacting:

Motorola, Inc.
6501 William Cannon Drive West
MD:OE17
Austin, Texas 78735
Phone (800) 521-6274
Fax (602) 437-1858
CRC@CRC.email.sps.mot.com

How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217, 1–800–441–2447 or 1-303-675-2140.
Customer Focus Center: 1–800–521–6274
JAPAN: Motorola Japan Ltd.; SPD, Strategic Planning Office, 141, 4–32–1, Nishi–Gotanda, Shinagawa–ku, Tokyo, Japan, 03–5487–8488
ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd., Silicon Harbour Centre, 2 Dal King Street, Tai Po Industrial Estate, Tai Po, New Territories, Hong Kong, 852–26668334
Mfax™, Motorola Fax Back System: RMFAX0@email.sps.mot.com; http://sps.motorola.com/mfax/; TOUCHTONE, 1-602-244–8609;
US & Canada ONLY, 1–800–774–1848
HOME PAGE: http://motorola.com/sps/

MOTOROLA
Semiconductor Products Sector